M335 L 1-4

THE OPEN UNIVERSITY
Mathematics: A Third Level Course
M335 Studies in Pure Mathematics

STUDIES IN PURE MATHEMATICS

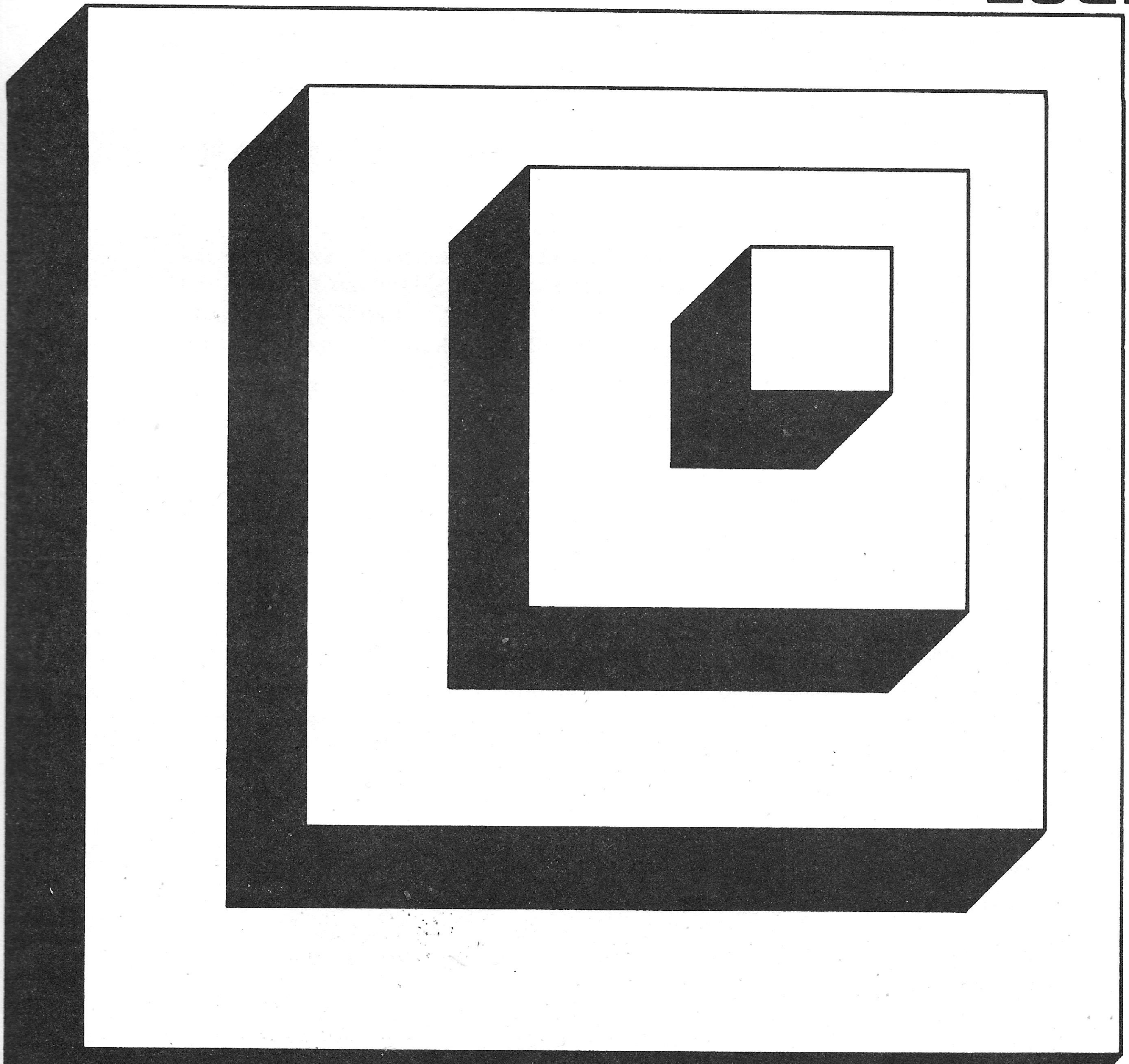
LOGIC

UNITS 1-4

THE OPEN UNIVERSITY
Mathematics: A Third Level Course
M335 Studies in Pure Mathematics
Prepared by the Course Team

LOGIC

UNITS 1-4



To do this option it is essential that you have a copy of the set book: *Computability and Logic* by G. Boolos and R. Jeffrey (Cambridge University Press, 1980, 2nd edition). In the text, this book will be referred to as **BJ**. A Turing machine kit is provided for use with *Unit 1*. You will need access to an audio-cassette player as there are audio-tapes accompanying the units.

It is hoped that you will find time to attempt all the problems in the units. If however, you are pressed for time, make sure to read our solutions to the problems, as these often contain teaching material. The objectives of each unit are stated purely in terms of what material might be tested in the assessment of the course, and can be found as the last section of each unit.

M335 Logic Option Course Team

Andrew Brown
Graham Flegg
Derek Goldrei
Alison Hughes
Ros Porter
Alan Slomson (University of Leeds)
Alex Wilkie (University of Oxford)
Hossein Zand

The Open University Press, Walton Hall, Milton Keynes.

First published 1981. Reprinted 1984, 1990, 1996, 1998

Copyright © 1981 The Open University

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publishers.

Designed by the Graphic Design Group of the Open University.

Produced in Great Britain by
Bell & Bain Ltd, Glasgow.

ISBN 0 335 14012 2

This text forms part of the correspondence element of an Open University Third Level Course.

For general availability of supporting material referred to in this text, please write to Open University Educational Enterprises Limited, 12 Cofferidge Close, Stony Stratford, Milton Keynes MK11 1BY, Great Britain.

Further information on Open University courses may be obtained from: The Admissions Office, The Open University, P.O. Box 48, Milton Keynes MK7 6AB

1.5

Contents

Unit 1 Turing Machines and Computability

1.0 Introduction to the Course **6**
1.1 Enumerability **6**
1.2 What is an Algorithm? **8**
1.3 Turing Machines **9**
1.4 Objectives of Unit 1 **24**
1.5 Appendix: Further Exercises **24**

Unit 2 Abacus Machines

2.0 Introduction **28**
2.1 The Busy Beaver Problem **28**
2.2 Introduction to Abacus Machines **33**
2.3 Abacus Computable Functions are Turing Computable **50**
2.4 Objectives of Unit 2 **56**

Unit 3 Recursive Functions

3.0 Introduction **58**
3.1 Primitive Recursive Functions **60**
3.2 Primitive Recursive Functions are Abacus Computable **67**
3.3 Minimization **70**
3.4 Objectives of Unit 3 **73**

Unit 4 Church's Thesis

4.0 Introduction **76**
4.1 Further Properties and Examples of Recursive Functions **76**
4.2 Recursive Functions Which Describe Turing Machines **82**
4.3 Turing Computable Functions are Recursive **88**
4.4 Church's Thesis **91**
4.5 Objectives of Unit 4 **92**

Index **93**

UNIT 1
TURING MACHINES AND COMPUTABILITY

1.0 Introduction to the Course

As the title of the set book suggests, we are going to be studying two main topics. *Computability* deals with fundamental theoretical questions about computers and algorithms. *Logic* deals with the principles of sound methods of reasoning. In studying these topics we shall meet ideas that have a very wide range of applications—in computing science, philosophy and linguistics, as well as many branches of pure mathematics. But our study of them is directed towards one central goal: Kurt Gödel's Incompleteness Theorem of 1930.

We have provided on the cassette tape for this section a brief historical introduction which traces some of the ideas which led to Gödel's Theorem, and the problems to which it gave an answer. *The material in this section is not assessed,* but we hope you will find it interesting. We suggest that you play the cassette tape now, and sit back and listen to it with your feet up, without worrying if you do not follow all the details. After all, this may be your last chance to relax until you complete the course!

START THE TAPE WHEN YOU ARE READY

For the sake of future reference, we record here the two questions raised in the introduction on the cassette and to which Gödel's Theorem will eventually provide answers.

Leibniz's Question

Is there an algorithm for deciding which statements of number theory are true?

Hilbert's Question

Can the consistency of number theory be proved using only non-dubious principles of finitary reasoning?

To provide an answer to Leibniz's question, we need a mathematical theory of algorithms, and this is provided in *Units 1–4*. Logic proper does not come in until the second half of the course. The set book assumes that the reader is already familiar with the basis of formal logic. We do not make this assumption, and so *Units 5* and *6* and much of *Unit 7* are designed to give you the logical information you need to understand the proof of Gödel's Theorem and its interesting consequences, which we come to in *Unit 8*.

1.1 Enumerability

We begin the mathematics of the course with a short section on enumerability, since the ideas in it will be useful later on. Before the first reading passage, we recall some definitions and notation you will have met in earlier courses.

We use the usual 'curly brackets' notation for sets. This notation is used in two ways. For finite sets, we can list all the elements, for instance $\{1, 2, 3, 5, 9\}$, but for very large finite sets this method is not practical, and for infinite sets it is impossible. For these sets we use the alternative notation which specifies the property that objects must have to be in the set, for instance:

$$\{n \in N : 1 \leq n \leq 1000\}; \qquad \{n \in N : n \text{ is divisible by } 7\}.$$

Note that in this course we shall use the symbol N to denote the *set of natural numbers*, and, following the practice of mathematical logicians in general (and

BJ in particular), treat 0 as a natural number. (Note that this differs from some other Open University courses, e.g. M101 and M203—not to mention the Number Theory option of M335.) Thus N stands for the set $\{0, 1, 2, 3, \ldots\}$. We shall also use P to stand for the set $\{1, 2, 3, \ldots\}$ of all positive numbers.

SEE [illegible] [illegible] P[illegible]

A *function* is a rule which associates with each element of one set (the *domain* of the function) an element of a second set (the *codomain*). In much of the course we shall consider functions with both domain and codomain N, the natural numbers, but in *Unit 1* we shall consider mainly functions with domain P, the positive integers, and codomain N. We write $f: A \longrightarrow B$ to indicate that f is a function with domain A and codomain B, and $x \longmapsto f(x)$ to show that the rule of f associates $f(x)$ in B with x in A. The elements of A are called the *arguments* of f. If $a \in A$ and $f(a) = b$ then b is the *value* of f for the argument a. If you need a further reminder about these ideas you should consult either M100 *Unit 1* or M101 *Block 1 Unit 4*, or the essay on *Sets and Functions* from M203.

There is one respect in which we shall want to extend the terminology of these earlier courses. We shall need to consider such functions as $h: P \longrightarrow P$ defined by

$$h(n) = \begin{cases} \dfrac{(n+1)}{2}, & \text{if } n \text{ is odd,} \\ \text{undefined otherwise.} & \end{cases}$$

Since the values of h are not defined for arguments which are even, h is not strictly a function with domain P in the sense explained above. We call such a function h a *partial function* with domain P, i.e. its domain is a subset of P. We shall see later that partial functions correspond to computational procedures that sometimes do not terminate, and that they play an important role in the theory of computability.

It will often help us later to describe a function $f: A \longrightarrow B$ as *total* when its domain is genuinely all of A.

READ BJ page 1, line 1 to page 6, line 15.

NOTES

(i) *Page 1, line 1*
Many books use *countable* instead of *enumerable*.

(ii) *Page 5, line 15*
By a *finite string* of letters, **BJ** means simply a finite sequence of letters, so that, for example

abce *cake* *dada*

are four-letter strings.

(iii) *Page 5, lines −8 to −4*
If you find this paragraph rather obscure, do not bother with it; nothing further will rely on it. ▲

PROBLEM 1 In the listing of all finite strings of letters of the alphabet which **BJ** describes on page 5, the first 26 strings are $a, b, c, \ldots, x, y, z$, the 27th string is aa, the 28th ab and so on. What is the 1000th string in this list?

PROBLEM 2 **BJ** page 7, Exercise 1.2.

PROBLEM 3 **BJ** page 8, Exercise 1.4.

PROBLEM 4 **BJ** page 8, Exercise 1.5. [Hint: In the previous problem, we showed how we can list all the finite strings of pluses and minuses. The suggestion that **BJ** makes is that we should find a way of interpreting such strings as naming or describing certain sets of positive integers. For example, we can think of the string

$$+ + - - - + - +$$

as naming the set $\{1, 2, 6, 8\}$. If you can see how we have done this you will have found one way of solving this problem.]

SOLUTION 1 The list looks like this:

$$\overset{\text{one-letter strings}}{\underbrace{a, \ldots\ldots, z,}_{26}}\ \overset{\text{two-letter strings}}{\underbrace{aa, \ldots, az, ba, \ldots, bz, za, \ldots, zz,}_{26 \times 26 = 676}}\ \overset{\text{three-letter strings}}{\underbrace{aaa, \ldots, aaz, \ldots, akz,}_{11 \times 26 = 286}\ \underbrace{ala, \ldots, all,}_{12}}\ldots$$

and $26 + 676 + 286 + 12 = 1000$, so the solution is the string *all.* ■

SOLUTION 2 See **BJ** page 9, Solution 1.2. ■

SOLUTION 3 See **BJ** page 9, Solution 1.4. ■

SOLUTION 4 See **BJ** page 9, Solution 1.5. ■

Having shown that certain sets are enumerable, it is natural to ask whether there are sets which are not enumerable. This question is answered in the affirmative in **BJ** Chapter 2, which is not, however, needed in the rest of this course.

1.2 What is an Algorithm?

Recall from Section 1.0 that by Leibniz's Question we mean: *Is there an algorithm for deciding which statements of number theory are true?*

In order to tackle this question, we shall need to make precise what we mean by an *algorithm.* We are all familiar with examples of algorithms from elementary mathematics. For example, there is the method of 'long multiplication' for doing a calculation such as 874 × 345. Anyone who knows the method can do this calculation in a routine or mechanical way. All we have to do is to follow a procedure consisting of a sequence of simple instructions. There is no need for any deep thought. This is typical of all algorithms. They are processes which produce an answer after a finite sequence of simple steps which are carried out according to specific instructions. We shall call a function *algorithmically computable* if there is an algorithm which computes the values of the function.

Now, an algorithm is just the sort of process we can envisage being done by a machine, and so we shall explain what we are going to mean by an algorithm by describing a simple kind of calculating machine. We shall then define an algorithm to be any process which can be carried out by this machine.

The sort of simple calculating machine which we describe was invented by the British mathematician Alan Turing (1912–1954) and is called a *Turing machine* in his honour. We describe how Turing machines work in Section 1.3.

It will be clear that a Turing machine is a purely mechanical device, and that any process which can be carried out by a Turing machine will fit in with our intuitive idea of an algorithm. At first sight, however, it might seem that the scope of a Turing machine is rather limited and that there are processes which we would want to count as algorithms which cannot be carried out by Turing machines. The surprising thing is that this first impression is wrong. Turing machines can carry out very complicated calculations. Indeed, there are good reasons for believing that any calculation which could be carried out by some mechanical computing device—such as, for example, a modern digital computer—could also be performed by a suitably programmed Turing machine.

If this is indeed true, then the precisely defined concept 'computable by a Turing machine' is equivalent to the intuitive idea of 'algorithmically computable'. The claim that these two concepts are equivalent is known as *Church's Thesis.* (This thesis is named after the American logician Alonzo Church who first formulated it in a paper read to the American Mathematical Society in 1935. To be accurate, Church stated the thesis as expressing the equivalence between the intuitive notion of an *effectively calculable* function and the precisely defined notion of a *recursive function.* We discuss the notion of a recursive function in *Unit 3* and prove that it is equivalent to that of a function whose values can be calculated by a Turing machine. Thus Church's formulation of his thesis is equivalent to our formulation.) A rigorous proof of this thesis is not possible since it equates a precise mathematical concept with an intuitive one.

In place of a proof, we can only present evidence in favour of the thesis. Part of the evidence is that other, seemingly more powerful, precise notions of algorithms have been proposed, but in each case it has been possible to show that any function which is algorithmically computable according to the apparently more powerful definition is also computable by a Turing machine. We give an example of this kind in *Unit 2* where we introduce another type of machine which Boolos and Jeffrey rather fancifully call an *abacus.* This sort of machine has random-access storage and is much more like a digital computer in its mode of working. However, we show that any calculation this seemingly more powerful machine could perform could also be done (albeit in a longer time) by a Turing machine.

1.3 Turing Machines

In this section, we aim to make precise which functions (or partial functions) are *computable.* (We say that a function is computable if there is an algorithm for computing the values of the function.)

We set about this by giving a precise mathematical definition of a class of functions called the *Turing computable functions.* The definition of these functions is precise in the sense that it uses only well-defined mathematical concepts and not intuitive ones such as that of an 'algorithm'. It will be clear from our definition that each Turing computable function is also a computable function in the intuitive sense. Conversely, we shall eventually accumulate sufficient evidence to make plausible the claim that each function which is computable in the intuitive sense is a Turing computable function. That is, we provide evidence for Church's Thesis, as explained in the previous section.

By a fortunate coincidence, it turns out that the notion of a Turing computable function also plays an important technical role in the proof of Gödel's Incompleteness Theorem. So the work in this and the succeeding three units is doubly valuable.

Our precise characterization of a *Turing computable function* will be in terms of the kind of machine we consider for calculating the values of such a function. Thus, schematically, a Turing computable function corresponds to the following diagram:

$$n \rightarrow \boxed{M} \rightarrow f(n)$$

Thus f is Turing computable if there is a Turing machine M which, given as input a natural number n, produces as output the corresponding value $f(n)$.

(This definition extends in a natural way to cover functions with more than one argument.) Our task is to specify what sort of operations the machine M can carry out. The mechanical (or electronic) aspects of M are not our concern. All that matters to us is what the machine can do.

READ BJ page 19, line 1 to page 22, line 21.

NOTES

(i) *Page 20, line 20*
Remember that we explained *Church's Thesis* in Section 1.2.

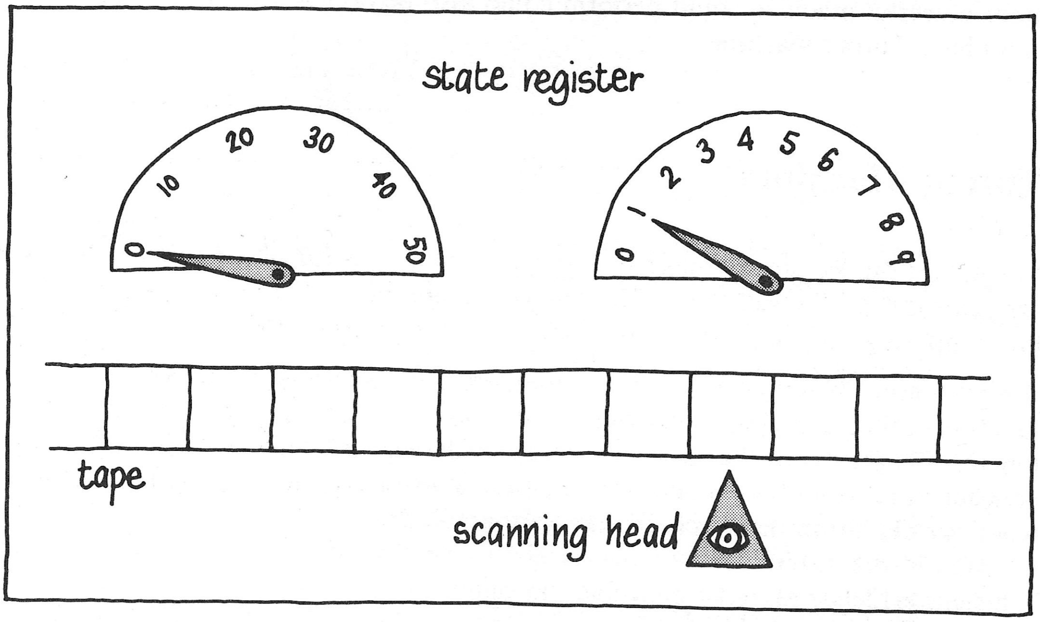
(ii) *Page 21, line 14*
'Ties' is the American word for 'railway sleepers'.

(iii) *Page 22, lines 17–21*
We explain the three different ways of specifying the program of a Turing machine in the tape section which follows. ▲

This tape section is designed to make you familiar with the way Turing machines work. Before starting the tape, make sure that you have the Turing machine kit in front of you.

START THE TAPE WHEN YOU ARE READY

1



2

(1) Halt the computation.

(2) Move the scanning head one square to the right.

(3) Move the scanning head one square to the left.

(4) Replace the scanned symbol by another symbol.

3

$$\overbrace{q_1 \qquad q_2 \qquad\qquad q_m}^{\text{machine states}}$$

state 1 state 2 state m

$$\overbrace{S_0 \quad S_1 \quad S_2 \qquad\qquad S_n}^{\text{machine symbols}}$$

informally

0 (or blank) 1 2 n

4

Machine table

		scanned symbol	
		S_0	S_1
	q_1	S_1q_1	Lq_2
state	q_2	S_1q_2	Lq_3
	q_3	S_1q_3	

S_1q_1 – replace the scanned symbol by S_1 and end up in state q_1

Lq_2 – move the scanning head one square to the left and end up in state q_2

5

5a

Set of quadruples

$q_1S_0S_1q_1$ $q_1S_1Lq_2$
$q_2S_0S_1q_2$ $q_2S_1Lq_3$
$q_3S_0S_1q_3$

5b

$q_iS_jAq_k$

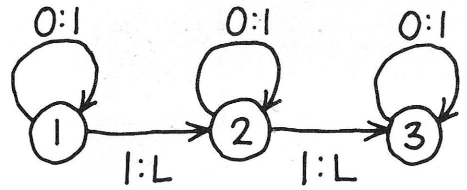
when in state q_i
scanning the symbol S_j
carry out action A
and end up in state q_k

These can be written in a line as

$q_1S_0S_1q_1; q_1S_1Lq_2; q_2S_0S_1q_2; q_2S_1Lq_3; q_3S_0S_1q_3.$

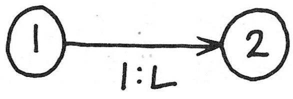
6

Flow graph



$q_1S_1Lq_2$

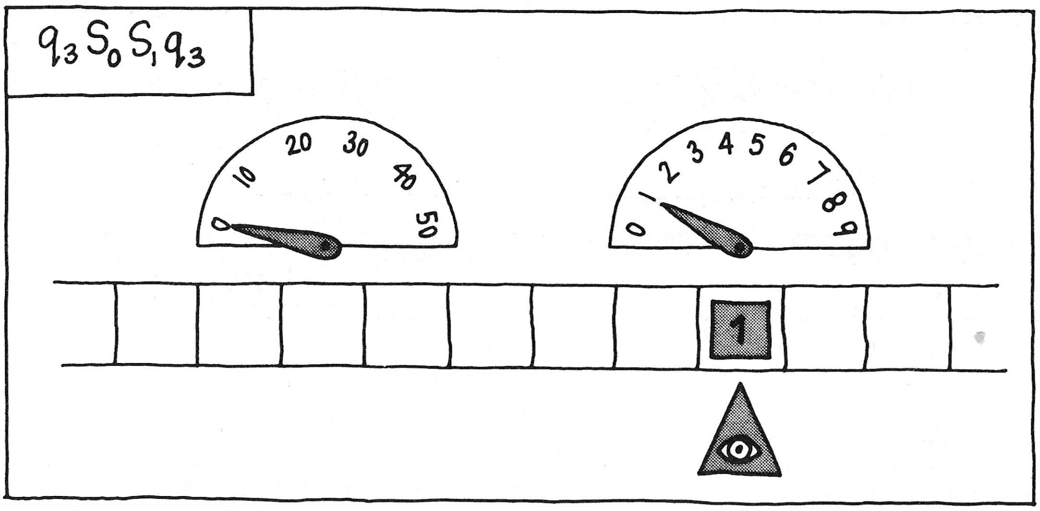
corresponds to the arrow



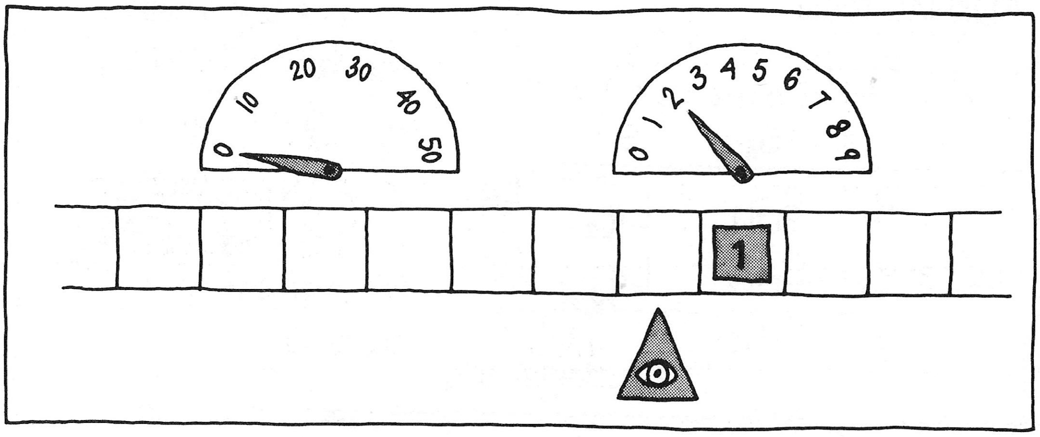
6a

$q_3 S_0 S_1 q_3$

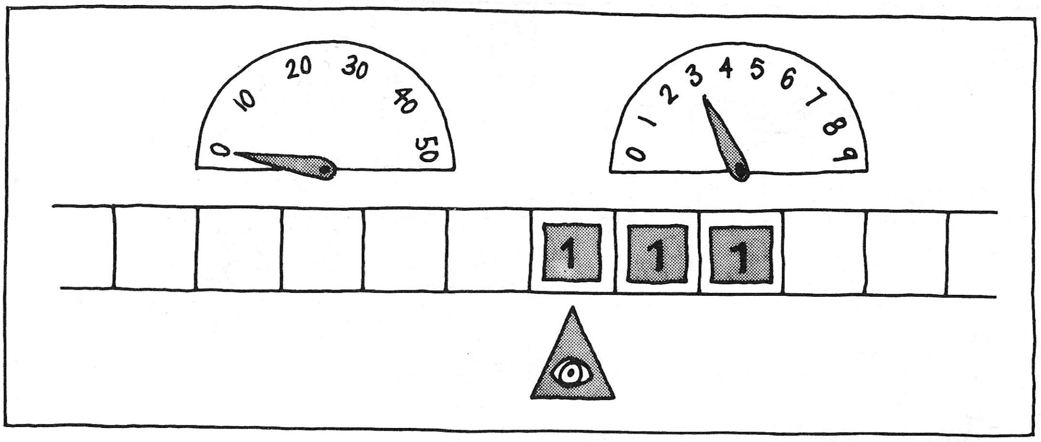
7


0
10
20
30
40
50
0
1
2
3
4
5
6
7
8
9
1

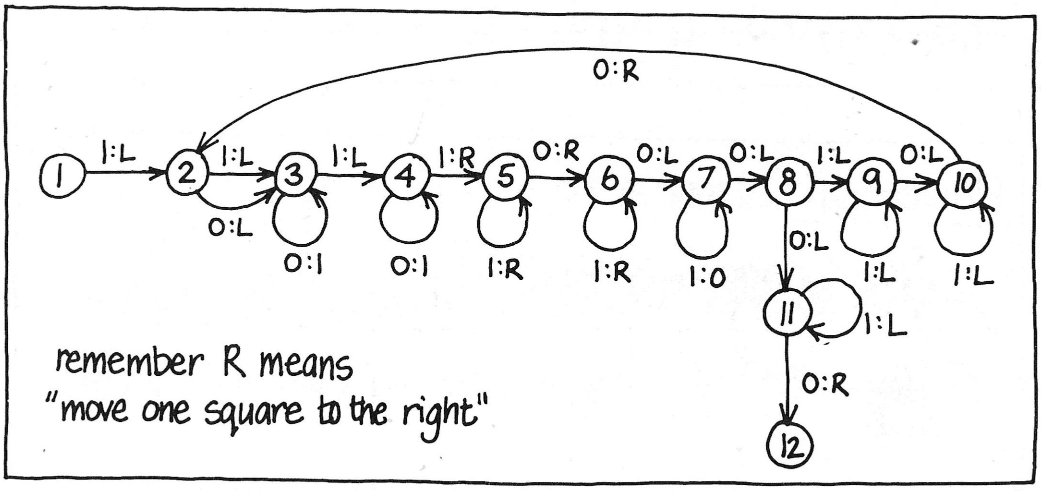
8


0
10
20
30
40
50
0
1
2
3
4
5
6
7
8
9
1

9

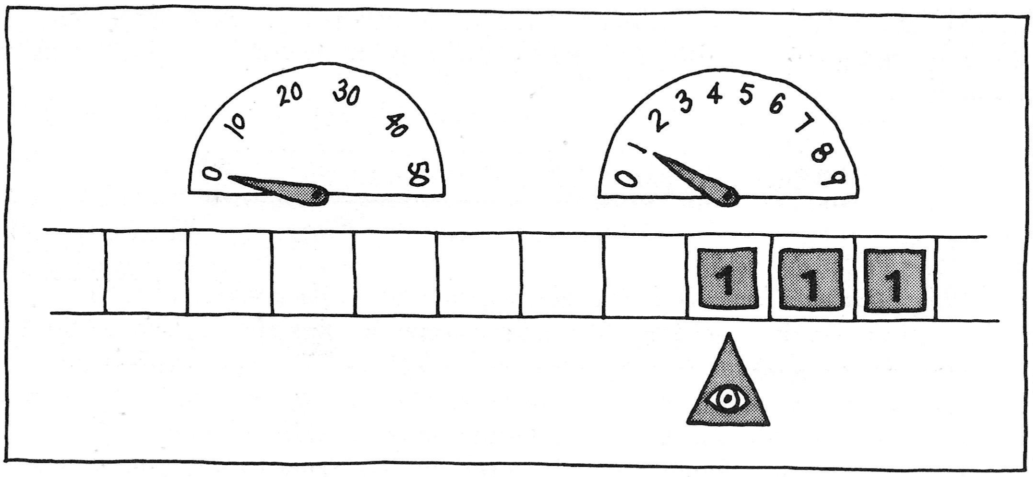

0
10
20
30
40
50
0
1
2
3
4
5
6
7
8
9
1
1
1

10

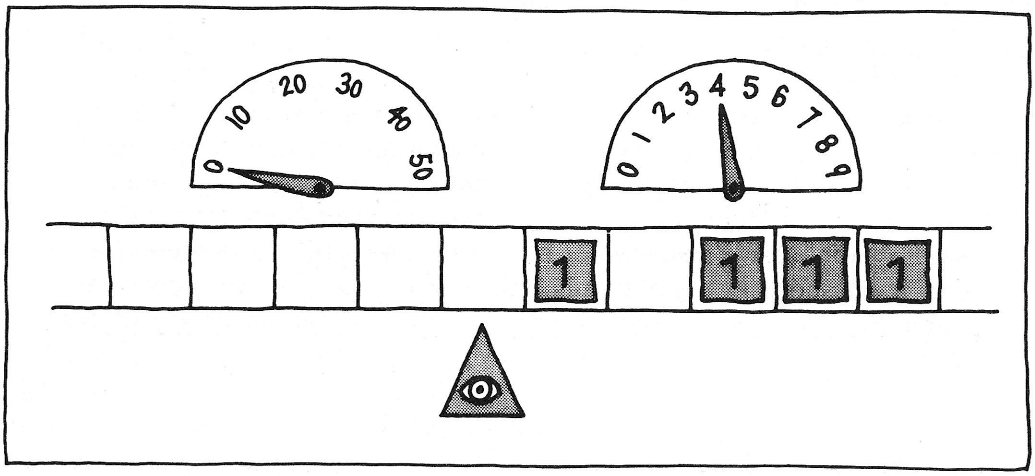

1
2
3
4
5
6
7
8
9
10
11
12
1:L
1:L
0:L
1:L
1:R
0:R
0:L
0:L
1:L
0:L
0:R
0:1
0:1
1:R
1:R
1:0
0:L
1:L
1:L
1:L
0:R

remember R means "move one square to the right"

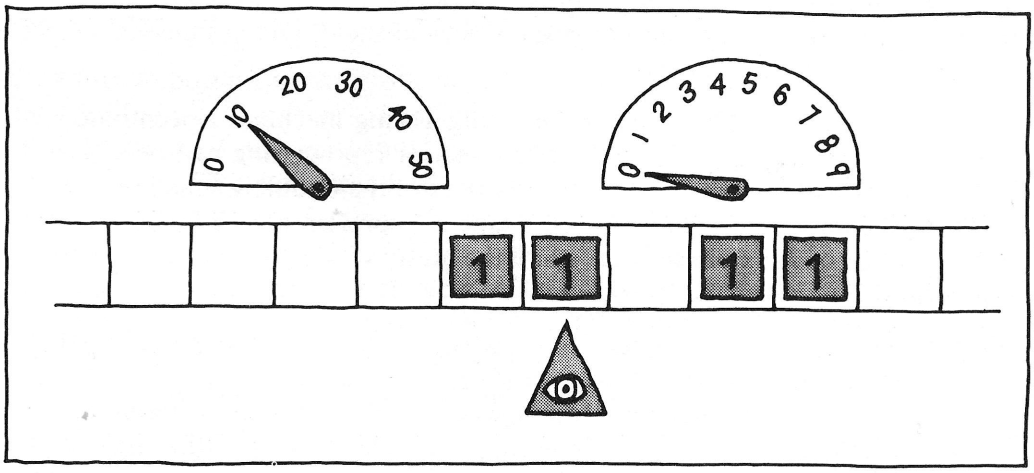
11


0 10 20 30 40 50
0 1 2 3 4 5 6 7 8 9
1 1 1

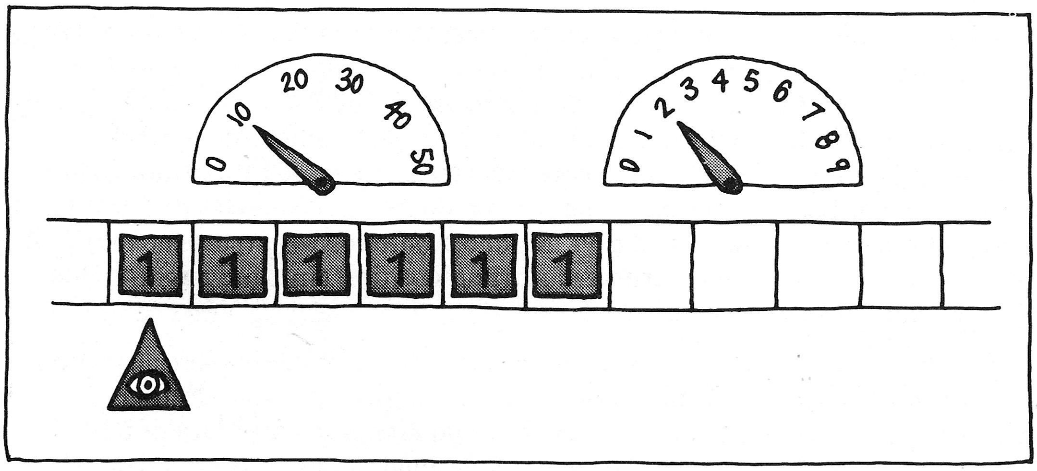
12


0 10 20 30 40 50
0 1 2 3 4 5 6 7 8 9
1 1 1 1

13


0 10 20 30 40 50
0 1 2 3 4 5 6 7 8 9
1 1 1 1

14


0 10 20 30 40 50
0 1 2 3 4 5 6 7 8 9
1 1 1 1 1 1

As you have in fact worked through Example 3.2 on the tape, we suggest that you now read the explanation of this example in **BJ** to accustom yourself to its notation.

READ BJ page 24, line 7 to page 25, line −12.

NOTES (i) *Page 24, line 8*

'*this machine*' Following Turing's original paper, **BJ** considers the program, i.e. the machine table or flow graph, as part of the machine. So it considers Example 3.2 as an example of a *different* Turing machine from that of Example 3.1. With our experience of digital computers, it is perhaps more natural to think of a single Turing machine which can be programmed in different ways. However, we shall stick to the terminology which **BJ** uses.

'*1s*' This just means 'ones'.

(ii) *Page 24, line 13*

The notation used here to show the situation of the machine should be clear. To save space, the squares on the tape are not shown, and the symbols on the relevant squares are printed in sequence. The number underneath one of the symbols is the state the machine is in, and the symbol above this number is the scanned symbol. Blank squares—corresponding to the symbol 0—at the ends of the tape are shown only when they are relevant. Thus to represent the situation in Frame 13 of the tape section, **BJ** would use

$$1\quad \underset{10}{1}\quad 0\quad 1\quad 1$$ ▲

Since the Turing machine in **BJ** Example 3.2 doubles the number of 1s on its tape, we can think of it as computing the values of the doubling function $n \longmapsto 2n$, for all positive integers n.

Before we can really begin to talk about using Turing machines to compute functions, we need to discuss the notation used for representing numbers. The machine of **BJ** Example 3.2 computes the values of the doubling function only if we regard the initial block of n 1s on the tape as representing the input n, and the block of $2n$ 1s on the tape when the computation halts as representing the output $2n$.

This notation in which the number n is represented by an unbroken string of n 1s is called *monadic notation.* Although for most purposes it is excessively cumbersome, it has considerable advantages of simplicity for the working of Turing machines and we shall continue to use it in most of the following examples.

The *input* is what is on the tape when the machine starts the computation. We assume that the machine always starts in state q_1 scanning the left-most 1 on the tape. We call this the *standard starting position.* The *output* is what is on the tape if and when the machine halts. In *Unit 2* we shall tighten this up by requiring that our machines stop, as well as start, in a standard position. Thus the machine of **BJ** Example 3.2 computes the doubling function because with input n (in monadic notation) it produces output $2n$ (in the same notation). If it starts in other than the standard starting position anything might happen, but we need not consider such cases.

Although this particular machine computes the values of the doubling function only when numbers are expressed in monadic notation, it would not be difficult to devise another machine which would compute the values of this function using, for example, the more usual decimal notation. (If you would like to see an example of a machine using decimal notation, look at **BJ**, page 26, Exercise 3.3.) In general, when it comes to the question of whether there is *some* Turing machine which computes the values of a particular function, it does not matter which notation it is proposed to use.

In the next example, we meet the important idea that two programs can be coupled together to produce a more complicated program. It is here that the flow graphs are particularly advantageous since they make it easier to visualize how programs can be joined.

READ BJ page 25, line −11 to page 26, line 7.

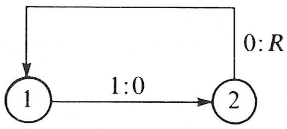
NOTES (i) *Page 25, lines −4, −3*
When **BJ** talks about 'identifying node 1 in Example 3.2 with node *n*...', it means that when the two flow graphs are coupled together, node 1 in Example 3.2 becomes the same node as node *n* of the machine which writes *n* 1s.

(ii) *Page 26*
Notice that in the flow graph, the states inside the dotted line have been given the same numbers as in the flow graph of Example 3.2. However, when the machines are coupled together, these states should be renumbered $n + 1, n + 2, \ldots, n + 11$, so that they have different numbers from the states used to write the *n* 1s. ▲

It is worth noting that this method of joining two machines works only because the machine that writes *n* 1s stops with the left-most of these 1s being scanned, which is the standard starting position for the second machine; i.e. the output of the first machine can be used directly as the input of the second. When we look further at joining machines together (in *Unit* 2), we shall often design machines that stop, as well as start, in a standard position.

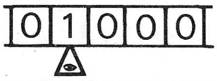
It is very important to do Problems 1 to 7 to consolidate your understanding of how Turing machines work. Problems 8 to 10 are harder, and if you are short of time you need only read through our solutions. If you require further practice in problems of the same standard as the TMA, you will find them in the Appendix to this unit, Section 1.5. Where you are asked to perform a specific calculation using a machine, we recommend that you start by using your Turing machine kit and at some stage get used to recording your calculation by writing down the sequence of configurations (as in **BJ** Example 3.2); our solutions will usually consist of this sequence. We also recommend that you represent the symbol 0 by a blank square on the tape of your kit.

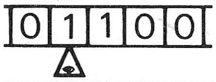
PROBLEM 1 Here is the flow graph of a Turing machine.

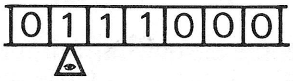


(i) Write down both the machine table and the set of quadruples corresponding to this flow graph.

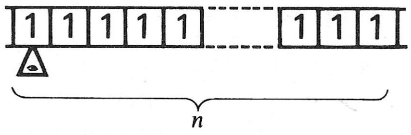
(ii) Investigate what happens when the machine is started in each of the following situations.

(a)


(b)


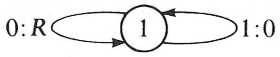
(c)


(iii) What do you think would happen if the machine started in the standard starting position scanning the left-most 1 of an unbroken string of n 1s, where n is a positive integer?



What function does this machine compute?

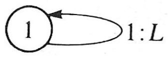
(iv) Would the same effect be obtained by using the Turing machine with the following flow graph?



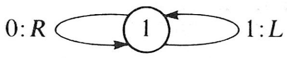
PROBLEM 2 We wish to design a Turing machine which, if started scanning the right-most of an unbroken string of n 1s, eventually halts when scanning the left-most 1 in the string. (A good practice when designing such a machine is to test it with some simple data.)

(i) Why are the following Turing machines unsuitable for the purpose described above? Illustrate your answer with the sequence of configurations obtained by using suitable test data.

(a)



(b)



(ii) Design a machine which does carry out the required task. Express your answer as a flow graph.

PROBLEM 3 Here is the machine table of a Turing machine.

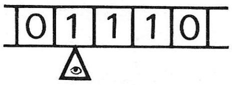
		Scanned symbol	
		0	1
State	q_1		$0q_4$
	q_2	$1q_6$	$0q_5$
	q_3	$1q_4$	$0q_6$
	q_4	Rq_2	Rq_2
	q_5	Rq_3	Lq_3
	q_6	Rq_1	

(i) Write down both the flow graph and the set of quadruples corresponding to this table.

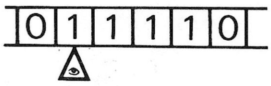
(ii) Which function does this machine compute?

[Hint: Try some particular calculations. We suggest that you work out what happens when the machine starts in each of the following situations in state 1. Here, and in subsequent examples, consider only what happens when the machine starts in the standard starting position.]

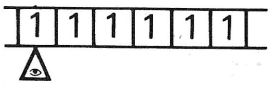
(a)



(b)

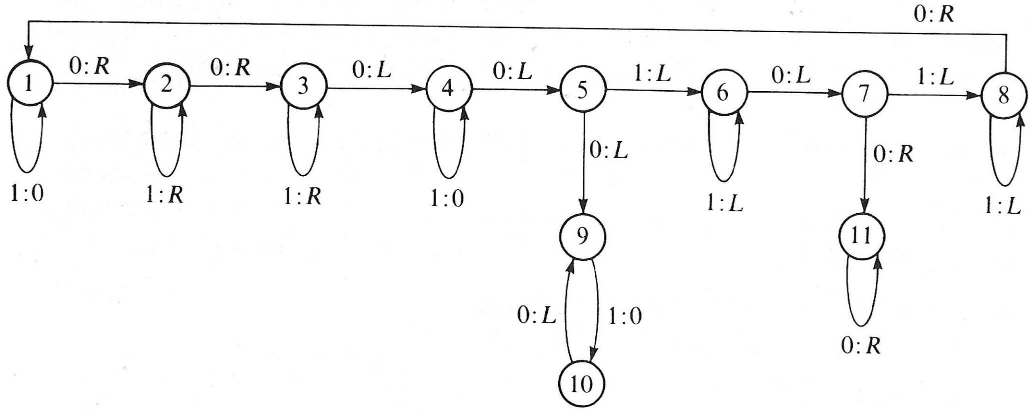


(c)

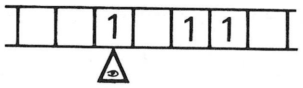


From working these examples it should become clear what output this machine produces when the input is n (expressed in monadic notation).

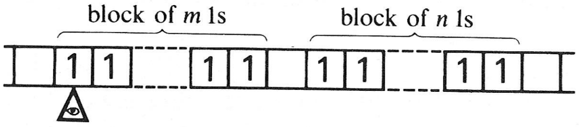
PROBLEM 4 Here is the flow graph of a Turing machine.



The input for this machine is a pair of positive integers, expressed using monadic notation as follows. We input, for example, the pair of numbers $(1, 2)$ in monadic notation as



and, in general, the *pair* (m, n) in *monadic notation* as



Note that the *standard starting position for a pair* (m, n) is with the scanning head beneath the left-most 1 of the string representing m.

Work out the output of this machine for the following inputs.

(i) $(1, 2)$ (ii) $(1, 3)$ (iii) $(2, 4)$ (iv) $(2, 1)$ (v) $(2, 2)$

For an input (m, n) the machine outputs $f(m, n)$. Try to find a simple description of the function f.

PROBLEM 5 Devise a Turing machine which, using monadic notation, inputs a pair (m, n) of positive integers in standard starting position (as described in Problem 4) and which halts scanning the left-most of a string of m 1s on an otherwise blank tape. Assume that the input (m, n) is also on an otherwise blank tape. The machine thus computes the function $(m, n) \longmapsto m$ for pairs (m, n) of positive integers.

COMMENT The next four problems are all concerned with designing Turing machines to carry out certain computations. Some of these problems are quite tricky. In each case, the best approach is first to decide how the Turing machine is to carry out the desired algorithm, and then to devise a set of machine instructions which will get the machine to operate in the desired way. A second look at Problems 3 and 4 above might be helpful.

PROBLEM 6 **BJ** page 26, Exercise 3.1. Note that this exercise is asking you to design a Turing machine which, using monadic notation, computes the *parity* function par which is defined by

$$\mathrm{par}(n) = \begin{cases} 0, & \text{if } n \text{ is even,} \\ 1, & \text{if } n \text{ is odd.} \end{cases}$$

PROBLEM 7 **BJ** page 26, Exercise 3.5, i.e. design a Turing machine which, using monadic notation, computes the addition function

$$(p, q) \longmapsto p + q.$$

The convention for inputting the pair of numbers (p, q) is that described in Problem 4 above.

COMMENT The next two problems are significantly harder than the previous two and so we have given you some hints on how to tackle them. They are both rather more difficult than (we hope) the assessment questions on this material will be, and so if you are short of time you may prefer simply to read the solutions.

PROBLEM 8 **BJ** page 26, Exercise 3.2. [Hint: We suggest that you get the machine to operate as follows:

(i) move from left to right until it finds a 1;
(ii) replace the 1 by a 3;
(iii) move back to the left end of the string;
(iv) move from left to right until it finds a 2;
(v) replace the 2 by a 3;
(vi) move back to the left end of the string;
(vii) replace the 3s at the extreme left end of the string by blanks and return to step (i);
(viii) if at step (i) the machine fails to find a 1, or at step (iv) fails to find a 2, it halts.

This machine thus deletes 1s and 2s in pairs and hence halts with a blank tape if and only if originally there was an equal number of these symbols on the tape.]

PROBLEM 9 **BJ** page 26, Exercise 3.4. This time you are being asked to devise a Turing machine to carry out multiplications in monadic notation. Again, this is rather tricky. **BJ**, with great ingenuity, gives an example of a multiplication machine which uses only the symbols 0 and 1. We think it is easier to use at least one additional symbol, say 2. The machine we have devised successively makes copies of the right-hand block of 1s using the left-hand block to keep track of how many copies it has made. It does this by replacing a 1 in the left-hand block by a 2 before making another copy of the right-hand block. Thus when there are no 1s remaining in the left-hand block the machine knows it has made the correct number of copies and is ready to halt the computation. We suggest that you try devising a machine to operate in a similar way.

COMMENT The problems above all go to show that certain functions can be computed by Turing machines. We say that a total function $f: P \longrightarrow N$ is *Turing computable* if there is a Turing machine which for each positive integer n as input in monadic notation halts with output $f(n)$ in monadic notation. Although Turing machines seem at first sight to be very simple, we hope that the above examples make it seem plausible that really complicated functions are Turing computable. The next unit will provide even more evidence for this. We conclude this section, however, with a problem which shows that not all functions are Turing computable and which enables us to deduce that there are some things which Turing machines cannot do. This problem is also somewhat harder than the assessment questions. Since it deals with an important theoretical result we suggest that if you are short of time you should simply read our solution.

PROBLEM 10 (i) Prove that the set of all Turing machines is enumerable.

[Hint: For the purpose of this question it is most helpful to think of a Turing machine as given by a set of quadruples.]

(ii) Deduce that the set of all Turing computable total functions $f: P \longrightarrow N$ is enumerable.

(iii) Deduce that there is a total function $g: P \longrightarrow N$ which is not Turing computable.

[Hint: Let $f_1, f_2, f_3, \ldots$ be an enumeration of all Turing computable total functions; such an enumeration exists by part (ii) above. Now consider the function g defined by

$$g(n) = f_n(n) + 1.$$

Obtain a contradiction from the assumption that g occurs somewhere in the list $f_1, f_2, f_3, \ldots$. It will follow that g does not occur in this list and so g is not Turing computable.]

SOLUTION 1 (i)

		Scanned symbol 0	Scanned symbol 1
State	q_1		$0q_2$
	q_2	Rq_1	

Quadruples: $q_1 1 0 q_2$, $q_2 0 R q_1$.

(ii) The successive configurations are as below.

(a)	(b)	(c)
$\underset{1}{1}\;0$	$\underset{1}{1}\;1\;0$	$\underset{1}{1}\;1\;1\;0$
$\underset{2}{0}\;0$	$\underset{2}{0}\;1\;0$	$\underset{2}{0}\;1\;1\;0$
$0\;\underset{1}{0}$	$0\;\underset{1}{1}\;0$	$0\;\underset{1}{1}\;1\;0$
	$0\;\underset{2}{0}\;0$	$0\;\underset{2}{0}\;1\;0$
	$0\;0\;\underset{1}{0}$	$0\;0\;\underset{1}{1}\;0$
		$0\;0\;\underset{2}{0}\;0$
		$0\;0\;0\;\underset{1}{0}$

(iii) If the machine is scanning a 1, it replaces it by a 0 and moves one square to the right, where it repeats this process until it scans a 0 while in state q_1. It then stops. Thus if it starts scanning the left-most of an unbroken string of n 1s, it will move to the right replacing these 1s by 0s and then stop one square further to the right. The machine computes the function with rule $n \longmapsto 0$ and domain the positive integers.

(iv) The new machine would certainly replace a block of n 1s by 0s, but it would never stop. It would continue moving to the right along the tape for ever! ■

SOLUTION 2 (i) (a) This machine will indeed move the scanning head to the left of the string, but will halt with the head one square to the left of the left-most 1, which is further than required. For instance, starting with a string of two 1s, we get the following sequence of configurations:

$$0\ \ 1\ \ \underset{1}{1}$$

$$0\ \ \underset{1}{1}\ \ 1$$

$$\underset{1}{0}\ \ 1\ \ 1$$

(b) In this case, the scanning head will move to the left of the string, but will oscillate between scanning the left-most 1 of the string and the square immediately to its left. Thus the machine will not halt. For instance, starting with a string of two 1s, we get the following sequence of configurations:

$$0\ \ 1\ \ \underset{1}{1}$$

$$0\ \ \underset{1}{1}\ \ 1$$

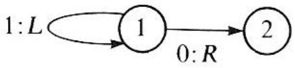
$$\underset{1}{0}\ \ 1\ \ 1$$

$$0\ \ \underset{1}{1}\ \ 1$$

$$\underset{1}{0}\ \ 1\ \ 1$$

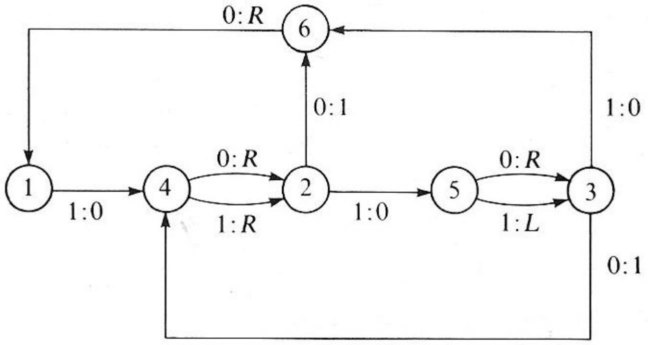
and so on ad infinitum.

(ii) One possible solution is given by the following flow graph.



■

SOLUTION 3 (i)



A different arrangement of the circles may give an apparently different, but equally correct, flow graph.

Quadruples: $q_1 1 0 q_4$, $q_2 0 1 q_6$, $q_2 1 0 q_5$, $q_3 0 1 q_4$, $q_3 1 0 q_6$, $q_4 0 R q_2$, $q_4 1 R q_2$, $q_5 0 R q_3$, $q_5 1 L q_3$, $q_6 0 R q_1$.

(ii) The machine computes the function f defined by

$f(n) =$ the remainder when n is divided by 3,

for each positive integer n. Note that when n is exactly divisible by 3, the machine will halt with a blank tape and we regard this as giving output 0.

The successive configurations for the sample calculations we suggested are:

(a)	(b)	(c)
$\underset{1}{1}\ 1\ 1\ 0$	$\underset{1}{1}\ 1\ 1\ 1\ 0$	$\underset{1}{1}\ 1\ 1\ 1\ 1\ 1\ 0$
$\underset{4}{0}\ 1\ 1\ 0$	$\underset{4}{0}\ 1\ 1\ 1\ 0$	$\underset{4}{0}\ 1\ 1\ 1\ 1\ 1\ 0$
$0\ \underset{2}{1}\ 1\ 0$	$0\ \underset{2}{1}\ 1\ 1\ 0$	$0\ \underset{2}{1}\ 1\ 1\ 1\ 1\ 0$
$0\ \underset{5}{0}\ 1\ 0$	$0\ \underset{5}{0}\ 1\ 1\ 0$	$0\ \underset{5}{0}\ 1\ 1\ 1\ 1\ 0$
$0\ 0\ \underset{3}{1}\ 0$	$0\ 0\ \underset{3}{1}\ 1\ 0$	$0\ 0\ \underset{3}{1}\ 1\ 1\ 1\ 0$
$0\ 0\ \underset{6}{0}\ 0$	$0\ 0\ \underset{6}{0}\ 1\ 0$	$0\ 0\ \underset{6}{0}\ 1\ 1\ 1\ 0$
$0\ 0\ 0\ \underset{1}{0}$	$0\ 0\ 0\ \underset{1}{1}\ 0$	$0\ 0\ 0\ \underset{1}{1}\ 1\ 1\ 0$
Input 3 Output 0	$0\ 0\ 0\ \underset{4}{0}\ 0$	⋮
	$0\ 0\ 0\ 0\ \underset{2}{0}$	as in part (a)
	$0\ 0\ 0\ 0\ \underset{6}{1}$	⋮ $0\ 0\ 0\ 0\ 0\ 0\ \underset{1}{0}$
	Input 4 Output 1	Input 6 Output 0

It can be seen that the machine erases 1s in cycles of three (via the route $q_1q_4q_2q_5q_3q_6q_1$). It reaches an originally blank square for the first time when in state q_1, q_2 or q_3, according as whether the remainder when n is divided by 3 is 0, 1 or 2. ■

SOLUTION 4 The following outputs are obtained:

(i) 1, (ii) 2, (iii) 2, (iv) 0, (v) 0.

The machine deletes 1s from the two blocks in turn until one block is exhausted. If this is the left-hand block, the machine halts scanning the left-most 1 of the right-hand block. If the right-hand block is exhausted first, the machine also deletes the remaining 1s in the left-hand block. Thus it computes the function f defined on the set of pairs of positive integers by

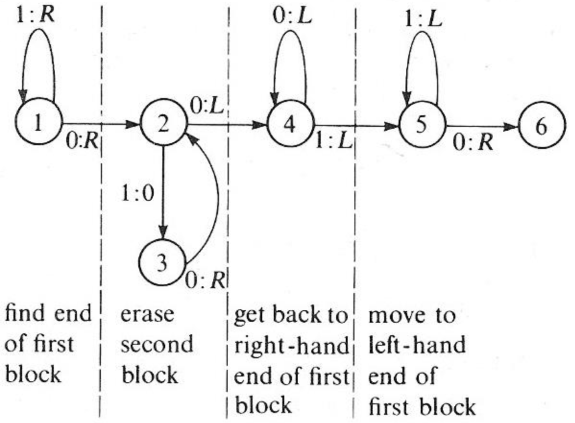
$$f(m,n) = \begin{cases} n - m, & \text{if } m \leq n, \\ 0, & \text{if } m > n. \end{cases}$$

We shall meet this function again later on. ■

SOLUTION 5 The problem here is to get the machine to recognize the second block of 1s, the block corresponding to n, to erase these, and to end up at the left-most end of the first block of 1s (corresponding to m).

The strategy of our solution is to move to the right of the first block, noting that when we eventually scan a 0 the first block of 1s has been completely scanned and the second block, corresponding to n, is about to begin on the next square to the right. We then replace all the 1s in the second block by 0s (as in Problem 1), and when we have done this, we return to the left until we hit the right-most of the first block of 1s. We then move to the left of this block (as in Problem 2).

One possible machine is thus


1:R
0:L
1:L
1
0:R
2
0:L
4
1:L
5
0:R
6
1:0
3
0:R
find end of first block
erase second block
get back to right-hand end of first block
move to left-hand end of first block

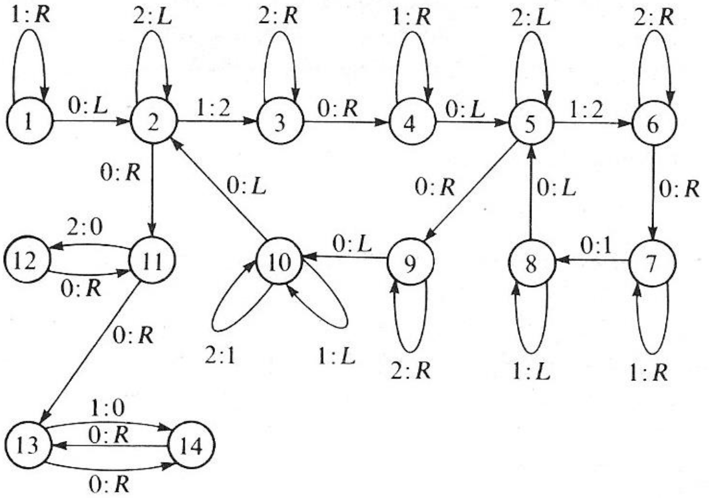
■

SOLUTION 6 See **BJ** page 28, Solution 3.1. Note that **BJ** has now switched to using B to denote a blank square. Of course, there are lots of other possible solutions and so your solution may well differ quite considerably from that given here. ■

SOLUTION 7 See **BJ** page 32, Solution 3.5. (There are lots of other valid solutions: for instance you could have closed the gap and removed the right-most 1.) The use of monadic notation makes this quite easy. ■

SOLUTION 8 See **BJ** page 29, Solution 3.2. Again, there are lots of alternative solutions. ■

SOLUTION 9 See **BJ** page 30, Solution 3.4. With its 'leap-frog' method **BJ** manages with only two symbols: B, representing the blank, and 1. We show the flow graph of a Turing machine which implements the method we suggested in our hint. Notice that although we have used more symbols than **BJ**, we have managed with fewer states.


1:R
2:L
2:R
1:R
2:L
2:R
1
0:L
2
1:2
3
0:R
4
0:L
5
1:2
6
0:R
0:L
0:R
0:L
0:R
2:0
12
11
0:R
10
0:L
9
8
0:1
7
0:R
2:1
1:L
2:R
1:L
1:R
1:0
13
0:R
14
0:R

■

SOLUTION 10 (i) Each Turing machine can be represented by a sequence of quadruples written down, in some order, with semicolons between them, as for example on **BJ** page 32, lines 3 and 4.

Viewed in this way, each Turing machine corresponds to a finite string of symbols taken from the following list of 15 symbols:

$SqLR0123456789;$

(where $0, 1, \ldots, 9$ are used to form the subscripts attached to S and q).

We saw in the section on enumerability, **BJ** page 5, that the set of all finite strings of letters of the alphabet is enumerable. In exactly the same way, the set of all finite strings of the above 15 symbols is enumerable. Not all of these finite strings correspond to Turing machines, but if we delete from the list all those that do not, we are left with an enumeration of all Turing machines, say M_1, M_2, $M_3, \ldots$. These machines therefore form an enumerable set.

(ii) Not every Turing machine computes the values of a total function $f: P \longrightarrow N$. To compute the values, in monadic notation, of a total function, the machine must, for each positive integer n as input, halt with some number as output. So it can fail to compute the values of a total function in two different ways. Firstly, for some input n the machine fails to halt. (We shall discuss this possibility in some detail in *Unit 2*.) Secondly, for some input n the machine does halt, but not with a block of 1s on its tape, so that the output is not a positive integer in monadic notation. However, if we delete from the list $M_1, M_2, M_3, \ldots$ those machines which do not compute the values of total functions, we are left with an enumeration, say $M'_1, M'_2, \ldots$, of all Turing machines which compute total functions.

Hence if f_n is the function computed by M'_n, then $f_1, f_2, f_3, \ldots$ is a list of all Turing computable total functions $f: P \longrightarrow N$. Since different machines may compute the same function, the list will contain repetitions, but that does not matter. Thus the set of all such functions is enumerable.

(iii) Define the function $g: P \longrightarrow N$ by $g(n) = f_n(n) + 1$, where f_n is the nth function in our enumeration of all Turing computable total functions. Clearly, g is a total function. We shall obtain a contradiction from the assumption that g is Turing computable.

If g is Turing computable, then it occurs somewhere in our list $f_1, f_2, f_3, \ldots$ of all Turing computable total functions. Say it is the Kth function in this list. Thus g and f_K are identical and so give the same value for all values of their arguments. That is, for each positive integer n,

$$g(n) = f_K(n).$$

So, in particular, this must be true for $n = K$. That is $g(K) = f_K(K)$.

Hence, using the definition of g, $f_K(K) + 1 = f_K(K)$.

Subtracting $f_K(K)$ from each side, we deduce that $1 = 0$,

which is a contradiction with the well-known fact that $1 \neq 0$. Thus the assumption that g is Turing computable leads to a contradiction. Therefore this assumption must be wrong. Hence g is not Turing computable. ■

The function g that we defined in Solution 10 is not Turing computable, but at first sight it seems that it is algorithmically computable in an intuitive sense, since the following might seem to be an algorithm for calculating the values of g. To calculate $g(n)$, we list all the Turing machine programs, crossing out those that do not compute total functions, until we reach the nth program, P'_n which does compute a total function. We now input n to this machine. The output will be $f_n(n)$, to which we add 1 to get the value of $g(n)$.

If this really were an algorithm for calculating the values of g, we should have an example of a computable function which is not Turing computable, thus contradicting Church's thesis. However, the method described above for calculating g is not an algorithm. The one step in the method which is not algorithmic is the point where we cross out those programs which do not compute total functions. To determine whether a program computes a total function we need to know its behaviour for *each* positive integer as input, and in general we cannot expect that there is a finite procedure for determining this. Indeed, if we accept Church's thesis, the above argument shows that *there is no algorithm to determine whether or not a Turing machine program computes a total function.*

This is our first example of an algorithmically undecidable mathematical question. We shall meet some more in *Unit 2*. Of course, this does not mean that we cannot in *particular cases* determine that a program does compute a total function, only that there is no algorithm which works in general.

If you are not happy about the use of Church's thesis in the above argument, we can avoid it as follows. We saw in Solution 10(i) that each Turing machine can be described by a finite string made up of the symbols needed for the quadruples which describe the machine. These symbols could be written on the tape of a Turing machine. Thus the description of a Turing machine in the form of its quadruples is in a suitable form for input into another Turing machine. So we can ask: *is there a Turing machine which, given as input the description of another Turing machine, T, halts with output 1 if T computes the values of a total function, and halts with output 0 otherwise*? We omit the technical details here, since they are rather messy, but it can be shown that if such a Turing machine existed, we could find a Turing machine which computed the function g and since we have proved that g is not Turing computable, it follows that *no such Turing machine exists.*

1.4 Objectives of Unit 1

Rather than attempting to summarize the unit, we list those things on which we may set assessment questions to test your understanding of the unit. After working through the unit, we feel you should be able to:

(i) demonstrate that sets are enumerable by showing how they can be listed in a single list;

(ii) translate a Turing machine table into a flow graph and translate a flow graph into a machine table;

(iii) given a machine table or a flow graph and a starting position, work out the subsequent behaviour of the Turing machine;

(iv) given a machine table or flow graph of a simple Turing machine, work out which function the machine computes;

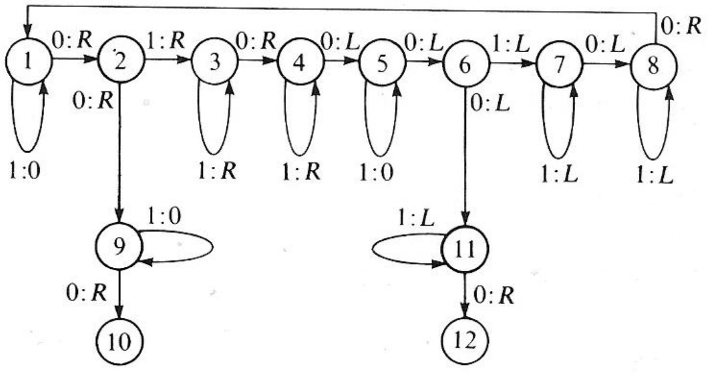
(v) given a simple function, devise a Turing machine which computes it.

We cannot say *exactly* what we mean by 'simple' in (iv) and (v) but a reasonable guide is that assessment questions which test (iv) will not be significantly harder than Problems 3 and 4 of Section 1.3, and questions which test (v) will not be significantly harder than Problems 6 and 7 of that section. Further practice, should you require it, in the skills required in (iv) and (v) is given by the exercises in the next section.

1.5 Appendix: Further Exercises

In case you feel that you need extra practice in problems of the same standard as the TMA questions, we have provided the following exercises.

EXERCISE 1 This exercise concerns the Turing machine with flow graph as below.



The machine accepts pairs (m, n) of positive integers expressed in monadic notation (as in Problem 4 of Section 1.3).

(i) Find out what is output by the machine for each of the following inputs.

(a) (3, 1)

(b) (1, 3)

(c) (2, 2)

(d) (4, 2)

(e) (2, 5)

(ii) What function on pairs of positive integers does this machine compute? Explain briefly how the machine works.

EXERCISE 2 Design a Turing machine which computes the function

$$n \longmapsto \text{remainder of } n \text{ on division by 5}, \quad \text{where } n \in P.$$

[This remainder should of course be in the set {0, 1, 2, 3, 4} represented by a string of 1s in monadic notation.]

Your machine should start in the standard position scanning the left-most of an unbroken string on n 1s, and halt scanning the left-most of the string of 1s representing the output.

SOLUTION 1 (i) In each case the output should be a block of 1s representing an integer in monadic notation. We give the sequences of configurations only for cases (a) and (b).

(a)

```
1  1  1  0  1  0
1
0  1  1  0  1  0
1
0  1  1  0  1  0
   2
0  1  1  0  1  0
      3
0  1  1  0  1  0
         3
0  1  1  0  1  0
            4
0  1  1  0  1  0
               4
0  1  1  0  1  0
            5
0  1  1  0  0  0
            5
0  1  1  0  0  0
         6
0  1  1  0  0  0
      11
0  1  1  0  0  0
   11
0  1  1  0  0  0
11
0  1  1  0  0  0
   12
```

Output 2

(b)

```
1  0  1  1  1  0
1
0  0  1  1  1  0
1
0  0  1  1  1  0
   2
0  0  1  1  1  0
      9
0  0  0  1  1  0
      9
0  0  0  1  1  0
         10
```

Output 2

(c) Blank tape output, i.e. 0

(d) Output 2

(e) Output 3

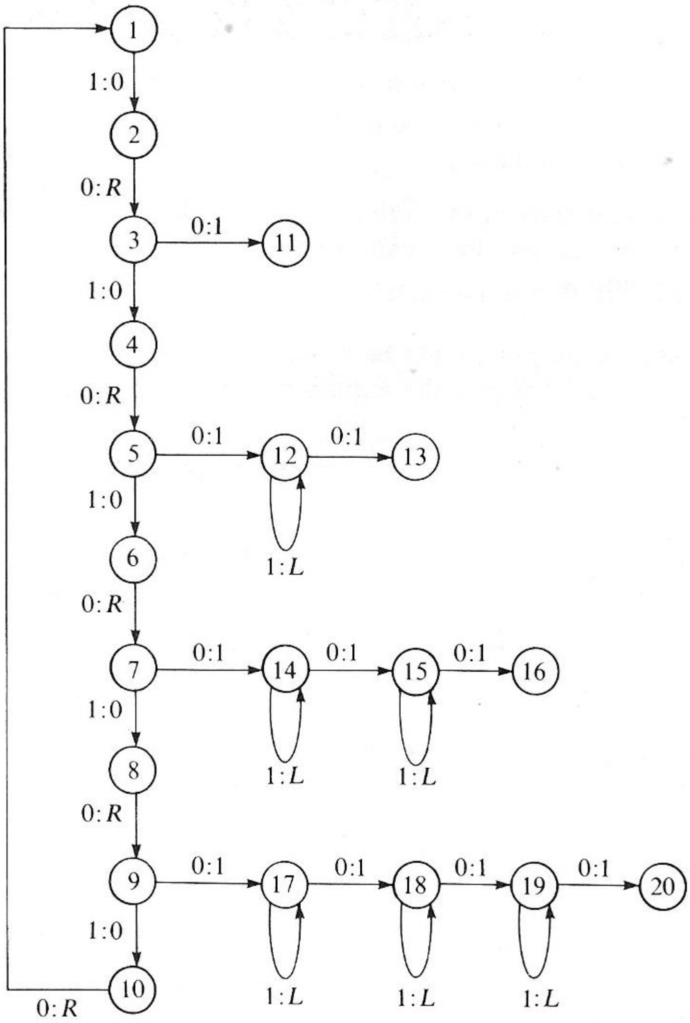
(ii) The machine computes the function

$$(m, n) \longmapsto |m - n|, \qquad \text{i.e.} \begin{cases} m - n, & \text{if } m \geq n, \\ n - m, & \text{if } m < n \end{cases}$$

for pairs of positive integers.

The machine works by repeatedly deleting a 1 from both the first string (initially with m 1s) and the second string (initially with n 1s). At the stage when one of the strings has had its last 1 removed, the machine enters an appropriate routine (at state 2 or state 6) to give the final answer and halt scanning the left-most 1 of the output string. ■

SOLUTION 2 One solution is as follows.



1
1:0
2
0:R
3
0:1
11
1:0
4
0:R
5
0:1
12
0:1
13
1:L
1:0
6
0:R
7
0:1
14
0:1
15
0:1
16
1:L
1:L
1:0
8
0:R
9
0:1
17
0:1
18
0:1
19
0:1
20
1:L
1:L
1:L
1:0
10
0:R

The machine will stop in states 1, 11, 13, 16 and 20 respectively as $f(n) = 0, 1, 2, 3$ and 4. The machine works by trying to delete five 1s from the string in states 1–10; if there are sufficient 1s in the string it repeats this process; otherwise it runs out of 1s to delete in one of the states 1, 3, 5, 7, 9, and depending on the case prints out the appropriate remainder. ■

UNIT 2
ABACUS MACHINES

2.0 Introduction

We begin the unit by continuing our study of Turing machines. In Problem 10 of Section 1.3 of *Unit 1* we gave an example of a function which cannot be computed by any Turing machine. In Section 2.1, by considering the so-called *Busy Beaver Problem*, we give another, more concrete, example of a function which is not Turing computable.

If the class of Turing computable functions were very small, the existence of these functions which are not Turing computable would not be very interesting. It would just show that Turing machines are not very powerful. However, in the remainder of this unit we show that this is not the case. In Section 2.2, we introduce a new type of calculating machine called an *abacus*. An abacus machine resembles a modern digital computer, and at first sight would seem to be much more powerful than a Turing machine.

The thing that makes a Turing machine work so slowly is that it can scan the squares on its tape only in the order in which they occur. You will have noticed from the examples in *Unit 1* that in a typical Turing machine calculation, very many of the steps are taken up with moving the scanning head from one part of the tape to another. By contrast, an abacus machine can have direct access to any of its *stores* (or *registers*, to use an alternative name) at any step of its calculation. In this respect, it is like a modern digital computer, in having what is usually called *random access storage* (or *memory*). ('Random' is a little misleading here since the sequence in which the machine uses its different registers is determined by its program and not by chance.)

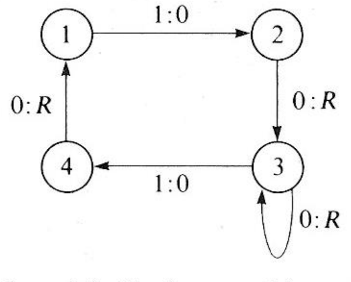
Because of their random access storage, abacus machines are easier to programme than Turing machines. We illustrate this with a number of examples, noticeably multiplication, which, as we saw in *Unit 1*, is rather complicated for a Turing machine.

Although abacus machines seem to be more powerful than Turing machines, we show in Section 2.3 that anything an abacus machine can do, a Turing machine can also do (although in a more cumbersome way). This shows that Turing machines are much more powerful than they appear to be at first sight. It provides our first piece of evidence that Turing machines are capable of carrying out any algorithmic calculation if suitably programmed. You will recall that the theory that this is the case is called Church's thesis, which we discussed in *Unit 1*.

2.1 The Busy Beaver Problem

In our solution to Problem 10 of Section 1.3 of *Unit 1*, we mentioned the possibility that for certain inputs a Turing machine computation might not halt. This is a real possibility, as shown by the following problems.

PROBLEM 1



Investigate what happens when this Turing machine starts in each of the following configurations: (i) $\underset{1}{1}\ 1$ (ii) $\underset{1}{1}\ 1\ 1$ (iii) $\underset{1}{1}\ 1\ 1\ 1$

(Recall that our notation means, for example in (i), that the machine starts in state 1 scanning the left-most of a block of two 1's on an otherwise blank tape.)

PROBLEM 2 What is the behaviour of the Turing machine whose machine table is given below, when it is started in state 1?

		Scanned symbol	
		0	1
State	q_1	$1q_1$	$0q_2$
	q_2	$0q_1$	

SOLUTION 1 In cases (i) and (iii) the machine halts with a blank tape, but in case (ii) it does not halt, but remains in state 3 moving the scanning head to the right for ever and ever. In general, with input n (i.e. with the machine started in state 1 scanning the left-most 1 of a block of n 1s), the machine halts with a blank tape if n is even, but does not halt if n is odd. ■

SOLUTION 2 Whatever the input, the machine does not halt. The scanning head remains in the starting position. The machine oscillates between states 1 and 2 and keeps on printing a 1 in the scanned square and then deleting it. ■

Turing machines which do not halt may still compute the values of partial functions. Thus the machine of Problem 1 computes the partial function f defined by

$$f(n) = \begin{cases} 0, & \text{if } n \text{ is even,} \\ \text{undefined otherwise.} \end{cases}$$

Because the Turing machines of Problems 1 and 2 were particularly simple, it was possible in each case to determine for which inputs the computation would halt. In general, however, this is not an easy problem. Consider the Turing machine given by the following machine table.

EXAMPLE 1

		Scanned symbol		
		0	1	2
State	q_1	Lq_3	Rq_2	Rq_1
	q_2	Lq_5	Rq_1	Rq_2
	q_3	Rq_1	Lq_4	Lq_3
	q_4		$2q_3$	Lq_4
	q_5		Lq_6	Lq_5
	q_6	Rq_{16}	Rq_7	Lq_6
	q_7		$2q_8$	Rq_7
	q_8	Rq_9		Rq_8
	q_9	$1q_{10}$	Rq_9	
	q_{10}	$1q_{11}$	Rq_{10}	
	q_{11}	$1q_{12}$	Rq_{11}	
	q_{12}	Lq_{13}	Lq_{12}	
	q_{13}	Rq_{14}	$2q_8$	Lq_{13}
	q_{14}	$1q_1$		$0q_{15}$
	q_{15}	Rq_{14}		
	q_{16}	Rq_{16}		$0q_{16}$

We can describe the behaviour of this machine as follows. Let

$$h(n) = \begin{cases} 3n + 1, & \text{if } n \text{ is odd,} \\ \dfrac{n}{2}, & \text{if } n \text{ is even.} \end{cases}$$

With input n, the machine computes successively the values

$$h(n),\ h(h(n)),\ h(h(h(n))), \ldots,$$

and halts when, and only when, it obtains the value 1. For example, with $n = 23$, the successive values are 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1 and the computation halts.

It is conjectured that whatever positive integer n you start with, this process always eventually produces the value 1. However, at the time of writing, this conjecture has been neither proved nor disproved. So we do not know whether this machine halts for every positive integer n as input. This raises the question as to whether there is an algorithm which will determine whether or not a given Turing machine with a particular input will halt. In this section we are going to prove that no such algorithm exists. □

The section of **BJ** that we now ask you to read comprises, in fact, the solutions to Exercises 3.7 and 3.8 of **BJ** page 27. We did not ask you to try these exercises as they are rather difficult and could be very time consuming.

READ BJ page 34, line 1 to page 41, line 10.

NOTES (i) *Page 34, lines 11–16*

This description of how Turing machines can be thought of as computing functions differs slightly from ours in *Unit 1*. **BJ** insists that the machine should not only halt with a block of 1s on the tape, but also that it should be scanning the left-most 1 in the block. We see later that this extra restriction is imposed so that the machine ends one computation in the standard starting position for another computation; this does not make a significant difference. For example, as we showed in Problem 2 of Section 1.3 of *Unit 1*, if a Turing machine is programmed to halt scanning the right-most 1 of a block, we can easily add a few more instructions to get it to halt scanning the left-most 1 instead.

(ii) *Page 34, lines -3, -2*

Thus, for example, the productivity of the Turing machine which computes the function f defined by $f(n) = 3n^2 + 5n + 7$ is 7, because $f(0) = 7$. The productivity of the Turing machine given in Problem 2 above is 0, since this machine computes a function which is not defined for $n = 0$.

(iii) *Page 35, line 7*

Notice first that in this chapter **BJ** is using 'B' to represent a blank square. It is important to realize that for each n there are only finitely many essentially different Turing machines with n states and which use just the symbols B and 1. We can assume that these states are labelled $q_1, \ldots, q_n$. The machine table of such a machine has $2n$ places for instructions (that is, 2 in each of the n rows). Each of these places can either be blank or contain an instruction of the form Aq_j. A can be any one of the 4 symbols B, 1, L, R; and q_j can be any one of the n symbols $q_1, \ldots, q_n$. So there are $4n$ possibilities for Aq_j. Therefore, including the case where there is no instruction, there are altogether $4n + 1$ possibilities for each place in the table. Since in drawing up the table there are $2n$ places to be filled, the total number of ways of filling these places is $(4n + 1)^{2n}$. Thus this is the number of different n-state Turing machines which use just the symbols B and 1. Each of these machines has a productivity, as defined by **BJ**. $p(n)$ is defined to be the largest of these $(4n + 1)^{2n}$ productivities. (If there were infinitely many essentially different machines with n states, their productivities might form an unbounded set, in which case there would be no largest productivity, and $p(n)$ could not be defined.)

(iv) *Pages 35–6, proof that $p(1) = 1$*

BJ says that there are infinitely many different one-state Turing machines because it is allowing the possibility that the one state is labelled q_1 or q_2 or .... Clearly, if two machines are exactly alike except that their states are differently labelled, they behave in exactly the same way. Thus nothing is lost if, as in Note (iii), we assume that the single state of a one-state machine is labelled q_1. With this stipulation, there are only 25 different one-state Turing machines which use only the symbols B and 1. This is the number given by the formula in Note (iii). To prove that $p(1) = 1$, it is necessary to prove that none of the 25 one-state Turing machines has productivity greater than 1, and that at least one of them has productivity 1. **BJ** first explains why all but one of the one-state machines have productivity 0 and then gives the example of the one-state machine which has productivity 1. This establishes the claim that $p(1) = 1$.

(v) *Page 37, line 5*
You will remember that we first met this idea of *plugging together* Turing machines in **BJ**, Example 3.3 on page 25. It is because we want to be able to plug machines together that we arrange for each Turing machine to end its calculation in the standard starting position for another calculation.

(vi) *Page 37, Figure 4-6*
As we have noted above, the number of different n-state Turing machines which use only the symbols B and 1 is finite. Each of these machines has a productivity, and $p(n)$ is the largest of these productivities. In general, more than one n-state machine might have productivity $p(n)$. By 'a most productive n-state machine' **BJ** means any one of the n-state Turing machines with productivity $p(n)$. This machine, when started with a blank tape, halts with output $p(n)$, that is, it halts scanning the left-most of a block of $p(n)$ 1s on a tape which is otherwise blank. The effect of the modification of this machine shown in Figure 4-6 is to get it to add one more 1 on the left-hand end of this block before halting. So the Turing machine in Figure 4-6 is an $(n + 1)$-state machine with productivity $p(n) + 1$. Therefore $p(n + 1)$, which is the productivity of the most productive $(n + 1)$-state machine, is at least $p(n) + 1$. That is, $p(n + 1) \geq p(n) + 1$, and this establishes the claim that $p(n + 1) > p(n)$.

(vii) *Page 38, Figure 4-8*
You may have noticed that the machines in this diagram are plugged together in a slightly different manner from previous examples. Don't worry about this now; we shall discuss it at the end of this section.

(viii) *Page 39, lines -6 to -1*
This form of Church's thesis may seem more implausible than the version we discussed in *Unit 1* since it says that 'algorithmically computable' is equivalent to 'computable by a Turing machine *which uses only the symbols B and 1*'. You might think that there are things which can be done by Turing machines but not by those which use only two symbols. The surprising thing is that this is not true: if a calculation can be done by some Turing machine which may use lots of symbols, it can also be done by another Turing machine which uses only the symbols B and 1.

(ix) *Page 40, lines 22–27*
We have not asked you to read Chapter 2 so this reference to Zeus may not make much sense. Fortunately nothing is lost if we ignore this paragraph. ▲

We can restate the conclusion which **BJ** draws on page 41, namely if Church's thesis is true, the halting problem is unsolvable in terms of Turing computability, as follows.

A Turing machine would be able to compute the productivity function p if it could carry out the following steps.

(a) List the machine tables, in the form of quadruples, of all the $(4n + 1)^{2n}$ n-state Turing machines which use only the symbols B and 1.

(b) For each machine table, decide whether the corresponding Turing machine would halt if started on a blank tape.

(c) In those cases where the machine would halt, carry out the computation performed by that machine when started on a blank tape and hence calculate the machine's productivity.

(d) Calculate the largest productivity obtained from all the n-state Turing machines in this way.

In fact, it is possible to design a Turing machine which would carry out steps (a), (c) and (d). This will become more evident from later work, but it should already be plausible that (a), which is a listing problem, and (d), which is a problem of calculating the largest of a given finite set of numbers, are within the scope of a Turing machine. Rather more thought is needed to show that (c) can also be done by a Turing machine, and we do not give a direct proof of this as the calculations needed are rather messy.

Thus since we know that p is not Turing computable, it must be step (b) which is beyond the power of a Turing machine. Hence we can conclude:

THEOREM 1 The halting problem for Turing machines started with blank tapes is not solvable by Turing machines.

Since we cannot use a Turing machine to solve the halting problem for Turing machines with the particular input of a blank tape, it follows even more so that:

THEOREM 2 The halting problem for Turing machines with arbitrary input is not solvable by Turing machines.

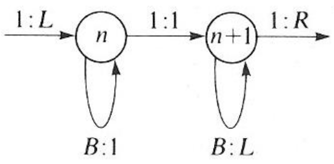
One reaction to these theorems might be that they simply show that Turing machines are not very powerful computational devices. But once we have finished giving the evidence for Church's thesis, by the end of *Unit 4*, we shall see that these theorems imply that the halting problem is not solvable by any computing device. (We emphasize that, of course, in particular cases it might be possible to decide if a given machine halts. The point is that there is no general algorithm for settling this question.) Since 'infinite cycling' is a real problem in trying to debug computer programs, this conclusion also has practical significance: it is not possible to programme a computer to systematically debug computer programs.

We end this section with an optional reading passage which takes up the point we mentioned in note (vii) above. If this point did not cause you any trouble, there is no need to read the next passage from **BJ**.

OPTIONAL: **READ BJ** page 41, line 14 to the end of page 42.

NOTE *Page 42, line 6*

You will see from Figure 4-8 (**BJ** page 38) that to avoid these conflicting instructions, instead of superimposing states n and $n + 1$, we join them with an arrow as shown below.



The symbols 1:1 alongside the arrow indicate that the machine follows this instruction when the scanned symbol is a 1, and that in this case the machine neither changes the scanned symbol nor moves the scanning head. Thus the function of the arrow is just to get the machine from state n to state $n + 1$ without any conflict in its instructions. ▲

2.2 Introduction to Abacus Machines

In this section we describe abacus machines and give examples of what they can do. In the next section we discuss the relation between abacus and Turing machines. Between them these sections cover Chapter 6 of **BJ**. This chapter begins with a list of some further changes in the standard form of a Turing machine computation. These are fairly minor and are introduced for convenience.

READ BJ page 52, line 1 to 'simple' on page 53, line 7.

NOTES (i) *Page 52, lines 4, 5*
The contrast here is between the way we would normally set out a multiplication sum on a two-dimensional page and the way a Turing machine works with a linear tape. Although the tape has width as well as depth, from the operational point of view it is one-dimensional since the scanning head can move only along the tape.

(ii) *Page 52, lines -10, -9*
Here by 'words on the alphabet $R, L, \ldots$' is meant 'finite strings of the symbols $R, L, \ldots$'. Our solution of **BJ** Exercise 3.4 did not meet this requirement since we used also the symbol 2. However, we did note that **BJ** gives a solution which uses just B and 1.

(iii) *Page 52, lines -8, -7*
Of course requirement (1) has to be modified in the obvious way for a function of one argument or a function of more than two arguments. ▲

We now come to our description of an abacus machine. The next reading passage gives a rough description of such a machine along with the operations which it can carry out. We follow the reading passage with some examples and problems which we hope will make these ideas clear before we go on to explain the notation used for abacus programs.

READ BJ page 54, line 5 to page 55, line -2.

NOTE *Page 55, lines 10–19*
Recall that **BJ** counts 0 as one of the natural numbers. In *Unit 1*, we usually considered functions with domain P, the set of positive integers, because there is a difficulty about representing 0 on the tape of a Turing machine if we wish to use only the symbols B and 1. The difficulty is that if we stick to the monadic notation of *Unit 1*, there is no way of distinguishing blank squares from the representation of the number 0 (since this would consist of a block of 0 1s, i.e. blank squares). The way we get round this is to use, from now on, a block of $(n+1)$ 1s to represent the number n. With this convention, 0 is represented by a single 1. If we adopt this new convention then, strictly speaking, we need to modify the Turing machines for computing various functions described in *Unit 1*. Doing so is a matter of rather tedious routine and so we shall avoid it. ▲

EXAMPLE 1 We give an example of a program using the basic instructions given by **BJ** on page 55, which has the effect of emptying box number n.

(1) Remove a stone from the pile in box number n, if there are any, and perform instruction number (1).

Clearly someone obeying this program will continue to remove one stone at a time from box number n until the box is empty. Since the program gives no instruction as to what to do when the box is empty, he will then stop.

EXAMPLE 2 Next we give an example of a program which has the effect of emptying the entire contents of box number m into box number n (where $m \neq n$).

(1) Remove a stone from the pile in box number m, if there are any, and perform instruction number (2).

(2) Add a stone to the pile in box number n and perform instruction number (1).

Carrying out this program has the effect of removing all the stones from box m and adding the same number of stones to box n. ☐

In both these examples, there was no instruction as to what to do in the case where the box from which stones were to be removed was empty. So in these cases both computations come to a halt. In more complicated programs, we may want to carry out some other instruction in this situation, as illustrated in the following problem.

PROBLEM 1 Consider this program.

(1) Remove a stone from the pile in box number m, if there are any, and perform instruction number (2); otherwise perform instruction number (3).

(2) Add a stone to the pile in box number p and perform instruction number (1).

(3) Remove a stone from the pile in box number n, if there are any, and perform instruction number (4).

(4) Add a stone to the pile in box number p and perform instruction number (3).

Assuming that m, n and p are all distinct, what is the effect of this program?

SOLUTION 1 In a fairly simple example like this, it might be possible to see what the program does just by looking at it. But to cope with more complicated examples we need to develop a method for doing sample calculations as we did with Turing machines. To do this all you need is a way of representing the contents of the different registers (i.e. boxes). If you have a real abacus, you can use the different rows of beads for this purpose. Or you could use matchboxes to represent the registers and matches for their contents. Probably the best way is to keep track of the calculation on a piece of paper, using different columns to represent the current content of each register, as in the table below. With this method you can also indicate the number of the instruction you are just about to carry out.

Trace table

Step number	**Instruction to be performed**	**Content of register** $[m]$	$[n]$	$[p]$
1	1	3	2	0
2	2	2	2	0
3	1	2	2	1
4	2	1	2	1
5	1	1	2	2
6	2	0	2	2
7	1	0	2	3
8	3	0	2	3
9	4	0	1	3
10	3	0	1	4
11	4	0	0	4
12	3	0	0	5

This table shows a sample calculation using the program of Problem 1, with the initial content of the registers as follows: $[m] = 3$, $[n] = 2$ and $[p] = 0$. For example, the table shows that at the sixth step, the machine performs instruction (2) with the following register contents: $[m] = 0$, $[n] = 2$ and $[p] = 2$. From the next line of the table, we see that the effect of carrying out this instruction is to increase the content of register p by 1, and that the next instruction to be carried out, at the seventh step, is instruction (1).

We call such a table the *trace table* of the computation. From it we see that if the machine starts with three stones in register m and two in register n, then it stops with five stones in register p, and the other two registers empty. Thus the effect is to transfer the contents of both registers m and n into register p. It is not difficult to see that the program always has this effect.

Register table

$[m]$	$[n]$	$[p]$
~~3~~	~~2~~	~~0~~
~~2~~	~~1~~	~~1~~
~~1~~	0	~~2~~
0		~~3~~
		~~4~~
		5

After some experience you should find that it is not necessary to write down the full trace table in order to keep track of a computation. It should be sufficient simply to record the current contents of each register and to change these records only when the register contents change. Thus at any step the bottom number in each column (the one not crossed out) shows you the current contents of the corresponding register. We call this abbreviated table the *register table* of the computation. We show the register table corresponding to the trace table given previously. ■

PROBLEM 2 Write down the trace table for the computation of the abacus machine of Example 2 when initially there are 2 stones in register m and 3 stones in register n.

PROBLEM 3 Write down what the register table would look like for the computation described in Problem 2

(i) after 3 steps of the computation (i.e. when you are about to carry out step 4), and

(ii) at the end of the computation.

SOLUTION 2

Step number	Instruction to be performed	Content of register $[m]$	$[n]$
1	1	2	3
2	2	1	3
3	1	1	4
4	2	0	4
5	1	0	5

■

SOLUTION 3 (i)

$[m]$	$[n]$
~~2~~	~~3~~
~~1~~	4
0	

(ii)

$[m]$	$[n]$
~~2~~	~~3~~
~~1~~	~~4~~
0	5

■

PROBLEM 4 Devise a program which has the effect of first emptying register n and then transferring the contents of register m to register n. (Assume $m \neq n$.)

PROBLEM 5 Devise a program which has the effect of interchanging the contents of registers *m* and *n*. (You will have to make use of a third register *p* as a temporary store for the contents of one of registers *m* and *n*: you should assume *p* is initially empty.)

SOLUTION 4 As with problems where you were asked to devise Turing machines, these problems have more than one solution, and so your programs may be different from ours. We have added annotations on the right to indicate how different bits of the program work.

(1) Remove a stone from box number *n* if possible and perform instruction (1); otherwise perform instruction (2).

This empties box *n*.

(2) Remove a stone from box number *m*, if possible, and perform instruction (3).

(3) Add a stone to box number *n* and perform instruction (2).

This transfers the contents of box *m* to box *n*. ■

SOLUTION 5 (1) Remove a stone from box number *m*, if possible, and perform instruction (2); otherwise perform instruction (3).

(2) Add a stone to box number *p* and perform instruction (1).

This transfers the contents of box *m* to box *p*.

(3) Remove a stone from box number *n*, if possible, and perform instruction (4); otherwise perform instruction (5).

(4) Add a stone to box number *m* and perform instruction (3).

This transfers the contents of box *n* to box *m*.

(5) Remove a stone from box number *p*, if possible, and perform instruction (6).

(6) Add a stone to box number *n* and perform instruction (5).

This transfers the contents of box *p* to box *n*. ■

This method for writing programs is rather cumbersome, so we now introduce a flow chart notation for abacus machine programs. This has two ingredients: circles, corresponding to the basic instructions, and arrows, indicating which instructions are to be carried out next. In Solutions 4 and 5 above, we found it convenient to bracket together parts of the program and add an explanation of what was the effect of the bracketed blocks. In the flow chart notation, we use a *block diagram* to indicate a piece of program which has a certain effect, without specifying the individual instructions.

READ BJ from the last line of page 55 to the end of page 57.

NOTES (i) *Page 56, Example 6.1*

The flow chart corresponds to the program we gave in our Example 1 above. The single circle represents the one instruction of this program. The arrow coming into this circle shows us where the program starts (i.e. which instruction the machine carries out first). Two arrows leave the circle. One tells us what to do next if it was possible to remove a stone from box *n*: in this case we have to go back and repeat instruction (1), so the arrow loops back into the circle from which it came. The other arrow tells us what to do next if box *n* was empty: in this program this arrow does not lead to another instruction, and this shows that in this situation the computation comes to a stop.

The block diagram on the right indicates the effect of the program as a whole. The effect is to leave 0 stones in box n, and this is shown by writing $0 \longrightarrow n$.

(ii) *Page 56, Example 6.2*
This flow chart corresponds to the program of our Example 2 above. This time there are two circles corresponding to the two instructions. So now the arrow at the top telling us which instruction is carried out first is really needed.

The block diagram on the right again indicates the effect of the program as a whole. We shall see from a later example that it is important to read this diagram from top to bottom. Box n ends up containing the number of stones equal to the sum of the numbers initially in box m and in box n. This is shown by $[m] + [n] \longrightarrow n$. And box m ends up empty. This is indicated by $0 \longrightarrow m$.

(iii) *Page 56, lines −10, −9*
If $m = n$ there are two different possibilities: either box m is initially empty, in which case the machine immediately halts; or box m initially contains at least one stone, in which case the machine cycles forever, alternately removing one stone from box m (instruction (1)) and then adding one stone to the same box (instruction (2)). So we have an example of the same phenomenon that we met with Turing machines: for some inputs the computation does not halt.

(iv) *Page 57, line 1*
Note that although **BJ** refers to '$[m] + [n] \longrightarrow n$' as a 'move', it corresponds not to a single step, but to several steps in the abacus machine computation.

(v) *Page 57, Example 6.3*
With the machine of the previous example, the computation ends with register m empty. In this example, we have a machine which has the effect of transferring the contents of register m to register n, but leaving register m at the end of the computation with the same contents as when it started. To do this, we make use of an additional register in a similar way to Solution 5 above.

(vi) *Page 57, lines −7 to −5*
We can record the effect of these 'moves' by a sample computation as follows:

	Contents of registers		
	$[m]$	$[n]$	$[p]$
Initially	3	2	5
After the 'move' $[m] + [n] \longrightarrow n$	3	5	5
After the 'move' $[m] + [p] \longrightarrow m$	8	5	5
After the 'move' $0 \longrightarrow p$	8	5	0

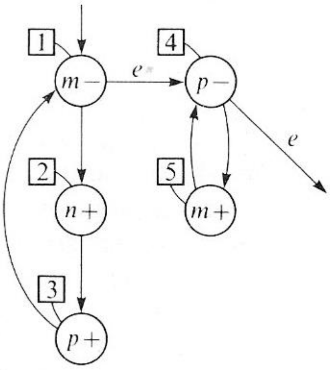
If we reverse the order of the last two 'moves' we get instead:

	Contents of registers		
	$[m]$	$[n]$	$[p]$
Initially	3	2	5
After the 'move' $[m] + [n] \longrightarrow n$	3	5	5
After the 'move' $0 \longrightarrow p$	3	5	0
After the 'move' $[m] + [p] \longrightarrow m$	3	5	0

The difference between the register contents at the end of these computations shows why the order of the 'moves' is important. ▲

PROBLEM 6 Write down the trace table for the computation of the abacus machine shown here (this is the machine of Example 6.3 to which we have added numbers to label the instructions), where the contents of the register are initially:

$$[m] = 2, \quad [n] = 3, \quad [p] = 0.$$



PROBLEM 7 Complete the following table:

	Contents of registers			
	$[m]$	$[n]$	$[p]$	$[q]$
Initially	1	2	1	3
After the 'move' $[m] + [n] \longrightarrow p$				
After the 'move' $[p] + [q] \longrightarrow q$				
After the 'move' $0 \longrightarrow q$				
After the 'move' $[n] \longrightarrow m$				

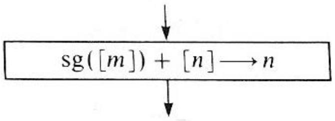
PROBLEM 8 Draw flow charts for the programs given in Solutions 4 and 5.

PROBLEM 9 Represent the programs given in Solutions 4 and 5 as block diagrams.

PROBLEM 10 The function sg (this abbreviates *signum*) plays an important role in *Unit 3*. It is defined by

$$\operatorname{sg}(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0. \end{cases}$$

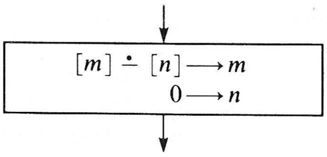
Draw a flow chart for a program which has the effect shown in the block diagram (i.e. if initially $[m] = 0$, then $[n]$ is unaltered, otherwise it is increased by 1).



PROBLEM 11 It is not difficult to devise an abacus machine program to do subtractions. The only snag is that since we are working with natural numbers, $x - y$ is defined only when $x \geq y$. So we use instead of ordinary subtraction the *difference* function, dif, defined by

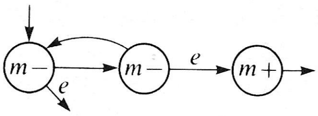
$$\mathrm{dif}(x, y) = \begin{cases} x - y, & \text{if } x \geq y, \\ 0, & \text{if } x < y. \end{cases}$$

Normally we write $x \dot{-} y$ instead of $\mathrm{dif}(x, y)$. The dot is there to warn us that we are using a modified form of subtraction. Draw a flow chart for a program whose effect is shown in the block diagram.

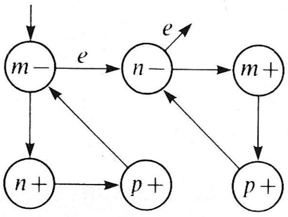


PROBLEM 12 Draw block diagrams corresponding to the flow charts given below.

(i)



(ii)



SOLUTION 6

Step number	Instruction to be performed	Contents of register $[m]$	$[n]$	$[p]$
1	1	2	3	0
2	2	1	3	0
3	3	1	4	0
4	1	1	4	1
5	2	0	4	1
6	3	0	5	1
7	1	0	5	2
8	4	0	5	2
9	5	0	5	1
10	4	1	5	1
11	5	1	5	0
12	4	2	5	0

■

SOLUTION 7

$[m]$	$[n]$	$[p]$	$[q]$
1	2	1	3
1	2	3	3
1	2	3	6
1	2	3	0
2	2	3	0

Note that if you got

1 2 4 3

for the second row of this table, then you probably made the mistake of carrying out the 'move'

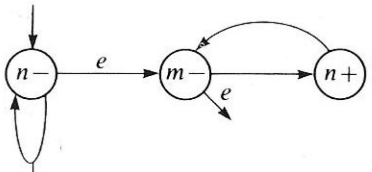
$$[m] + [n] + [p] \longrightarrow p$$

instead of

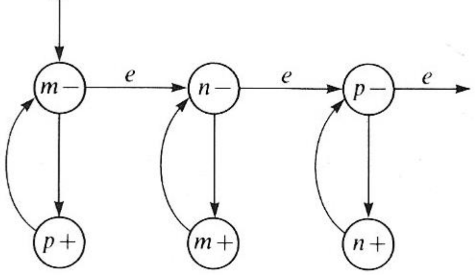
$$[m] + [n] \longrightarrow p.$$ ■

SOLUTION 8

Problem 4:

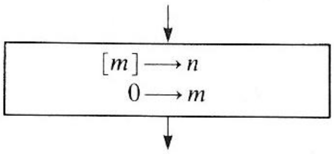


Problem 5:

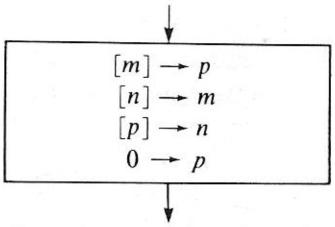


Of course, your solutions might look slightly different if you arranged the circles corresponding to the instructions in a different way. ■

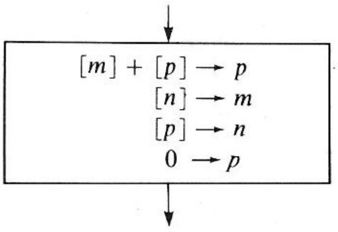
SOLUTION 9 *Problem 4:*


$[m] \longrightarrow n$
$0 \longrightarrow m$

Problem 5: If we assume that initially $[p] = 0$ then the block diagram is:

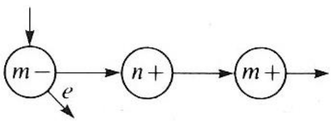

$[m] \rightarrow p$
$[n] \rightarrow m$
$[p] \rightarrow n$
$0 \rightarrow p$

but if we want to describe what the machine does in the general case where register number p need not be empty to start with, then we have:


$[m] + [p] \rightarrow p$
$[n] \rightarrow m$
$[p] \rightarrow n$
$0 \rightarrow p$

■

SOLUTION 10 To calculate $\text{sg}([m])$, the machine has to test the content of register m to see if it is 0. We use the '$m-$' instruction for this purpose. If $[m] = 0$, we do not alter $[n]$ but halt at once, while if $[m] \neq 0$ we increase $[n]$ by 1. In this case, carrying out the '$m-$' instruction has subtracted 1 from $[m]$ and so we have to add this back before halting. (Since the block diagram does not specify any change to $[m]$, its value at the end of the calculation must be the same as at the start.) In this way, we arrive at the following program:

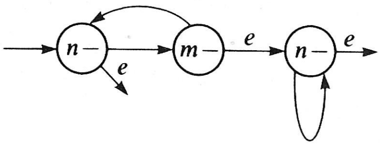


■

SOLUTION 11 We get the machine to successively subtract 1 from $[n]$ and $[m]$, using the '$n-$' and '$m-$' instructions. If initially $[m] \geq [n]$, $[n]$ will become 0 first, and since we have subtracted as many 1s from $[m]$ as from $[n]$, the content of register m will now be equal to

the initial value of $[m]$ − the initial value of $[n]$,

and so the machine can halt. If initially $[m] < [n]$, the content of register m will become 0 first. We shall then need to make $[n]$ equal to 0 before halting. Thus we are led to the following flow chart:

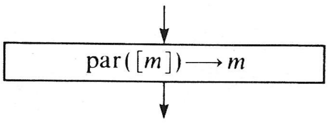


A few sample calculations should make it clear how this machine operates. ■

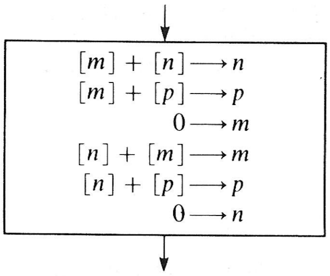
SOLUTION 12 You might be able to tell what these machines do just by looking at the flow charts, but probably you will need to do several sample computations to get the hang of things.

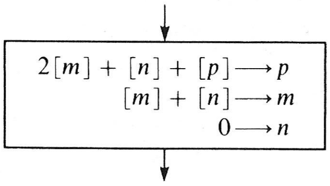
(i) If initially $[m]$ is even, the machine halts with $[m] = 0$; otherwise it halts with $[m] = 1$. We can sum up this behaviour by the block diagram shown. Remember that we introduced the function par in *Unit 1*, and that it is defined by

$$\operatorname{par}(x) = \begin{cases} 0, & \text{if } x \text{ is even,} \\ 1, & \text{if } x \text{ is odd.} \end{cases}$$



(ii) The effect of the left-hand loop in the program is to transfer the content of register m to both registers n and p. Then the right-hand loop has the effect of transferring the content of register n to both registers m and p. These 'moves' are indicated in the block diagram shown. However, this can be simplified; a few sample computations should convince you that the same effect can be achieved by carrying out the 'moves' shown in the second block diagram. In both cases the order of the 'moves' is very important. ■





We can regard the abacus machine of **BJ**, Example 6.3, page 57, as a machine which does additions. The next passage from **BJ** gives an example of an abacus machine which carries out multiplication. At this point we see the advantage of abacus machines over Turing machines. You will remember from *Unit 1* that it was rather difficult to devise a Turing machine to do multiplication. Devising an abacus machine which carries out multiplication is a lot easier. The machine uses the fact that multiplication can be regarded as repeated addition.

READ BJ page 58, line 1 to page 59, line −5.

NOTES (i) *Page 58, line 4*
Here '*de novo*' means 'from the begining', i.e. instead of designing the flow chart from scratch, we make use of the flow chart for addition already given in Example 6.3.

(ii) *Page 58, diagram*
The block diagram in the middle just tells us what the program is intended to do. Thus, provided registers n and p are initially empty, by the end of the computation, register n contains the product of the original contents of registers m_1 and m_2, and register m_1 is empty. Since nothing is specified about registers m_2 and p, the implication is that at the end of the

computation the content of these registers is the same as at the start of the computation.

The abbreviated flow chart on the left shows how we can make the structure of a program clearer by inserting a block diagram into the flow chart instead of the detailed program which carries out the computation described by the block diagram.

(iii) *Page 59, diagram*
In the abbreviated flow chart, 'cumulative multiplication' is just the name for the right-hand block diagram. You may wonder why this is so much more complicated than the multiplication program given in **BJ** Example 6.4. The reason is that, as you will see from the block diagram on page 58, the effect of that program is to erase one of the numbers being multiplied, but to perform exponentiation by repeated multiplication we must not lose either of the numbers that we multiply. This is the purpose of register q and the reason why the program keeps shifting the content of register n into register q and back again. The next problem is designed to help you get a feel for what this program does. ▲

PROBLEM 13 Investigate how the programs of **BJ** Examples 6.4 and 6.5 work by completing the following tables.

(i) Here we ask you to carry out the 'moves' given by the abbreviated flow chart and block diagram of **BJ** Example 6.4.

	$[m_1]$	$[m_2]$	$[n]$
Initially	3	4	0
(a) After 'move' $[m_1] - 1 \longrightarrow m_1$			
(b) After 'move' $[m_2] + [n] \longrightarrow n$			

Then repeat steps (a) and (b) until you reach (a) with $[m_1] = 0$, when the computation halts.

(ii) Here we ask you to carry out the 'moves' given by the flow chart of **BJ** Example 6.5.

	$[m_1]$	$[m_2]$	$[n]$	$[q]$
Initially	3	5	0	0
(a) After move $[n] + 1 \longrightarrow n$				
(b) After move $[m_1] - 1 \longrightarrow m_1$				
(c) After move $[n] \cdot [m_2] \longrightarrow q$				
(d) After move $0 \longrightarrow n$				
(e) After move $[q] + [n] \longrightarrow n$				
(f) After move $0 \longrightarrow q$				

Then repeat steps (b), (c), (d), (e) and (f) until step (b) is reached with $[m_1] = 0$, when the computation halts.

PROBLEM 14 Write out the full flow chart of the exponentiation program given in **BJ** Example 6.5.

SOLUTION 13 (i)

	$[m_1]$	$[m_2]$	$[n]$
Initially	3	4	0
After step (a)	2	4	0
After step (b)	2	4	4
After step (a)	1	4	4
After step (b)	1	4	8
After step (a)	0	4	8
After step (b)	0	4	12

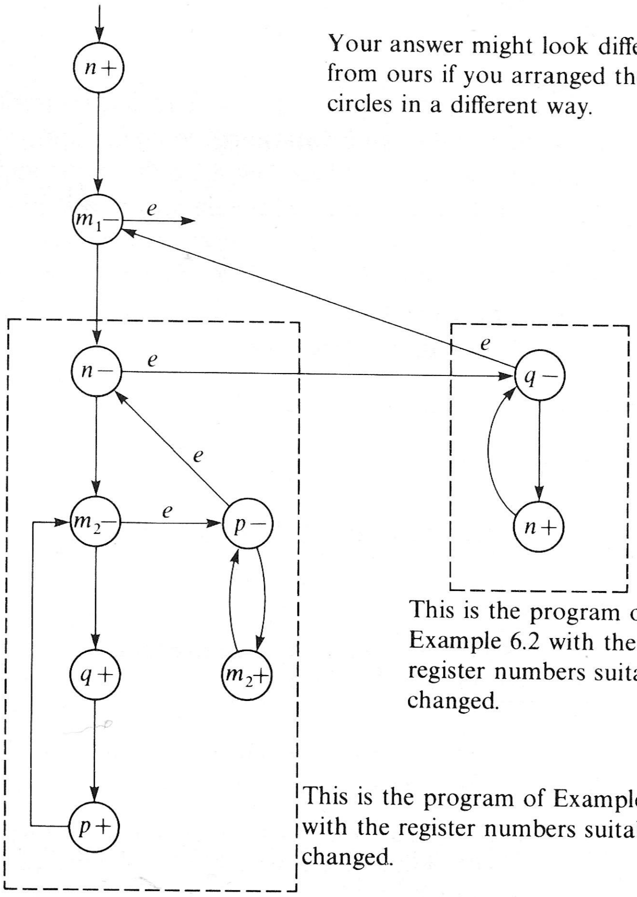
and the computation halts with 3 × 4 in register *n*.

(ii)

	$[m_1]$	$[m_2]$	$[n]$	$[q]$
Initially	3	5	0	0
After step (a)	3	5	1	0
After step (b)	2	5	1	0
After step (c)	2	5	1	5
After step (d)	2	5	0	5
After step (e)	2	5	5	5
After step (f)	2	5	5	0
After step (b)	1	5	5	0
After step (c)	1	5	5	25
After step (d)	1	5	0	25
After step (e)	1	5	25	25
After step (f)	1	5	25	0
After step (b)	0	5	25	0
After step (c)	0	5	25	125
After step (d)	0	5	0	125
After step (e)	0	5	125	125
After step (f)	0	5	125	0

and the computation halts with 5^3 in register *n*. ■

SOLUTION 14



Your answer might look different from ours if you arranged the circles in a different way.

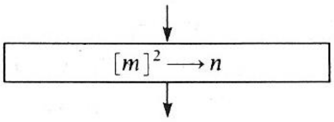
This is the program of Example 6.2 with the register numbers suitably changed.

This is the program of Example 6.4 with the register numbers suitably changed.

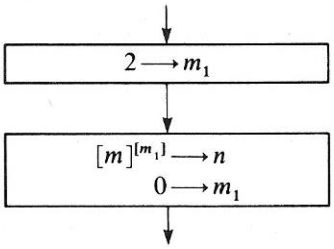
■

EXAMPLE 3 We explain how to devise a program for squaring numbers. As with all such examples more than one approach is possible, and what seems most natural to you might well not agree with what seems the best method to us.

We want to devise a program whose behaviour is summed up by the following block diagram.

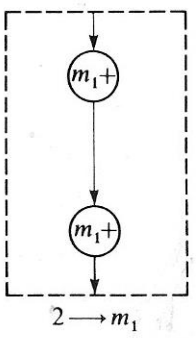

$[m]^2 \longrightarrow n$

To achieve this we can use the program for exponentiation given in **BJ** Example 6.5. All we have to add to this are some instructions at the beginning to make the exponent equal to 2, i.e. we have to set $[m_1] = 2$. Thus we get the following program.


$2 \longrightarrow m_1$
$[m]^{[m_1]} \longrightarrow n$
$0 \longrightarrow m_1$

This is achieved as on the right (provided that initially $[m_1] = 0$)

This is the program of Example 6.5 with 'm_2' replaced by 'm'.

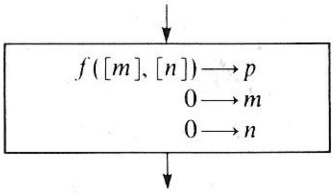

m_1+
m_1+
$2 \longrightarrow m_1$

There is no need to write out in full the flow chart corresponding to the exponentiation block since we already have a record of it elsewhere.

EXAMPLE 4 Next we show how to devise a program for comparing the size of numbers. That is, we shall devise a program which works in the following way. If initially $[m] \leq [n]$, the computation will halt with $[p] = 0$, while if initially $[m] > [n]$, it will halt with $[p] = 1$. (In both cases we shall have $[m] = [n] = 0$ when the computation halts.) Thus if we define the function f by

$$f(x, y) = \begin{cases} 0, & \text{if } x \leq y, \\ 1, & \text{if } x > y, \end{cases}$$

then the behaviour of this program can be summed up by the block diagram shown.


$f([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$

To achieve this we note that

$$x \leq y \iff x \dot{-} y = 0 \iff \mathrm{sg}(x \dot{-} y) = 0$$

and

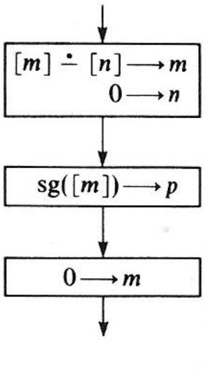
$$x > y \iff x \dot{-} y \neq 0 \iff \mathrm{sg}(x \dot{-} y) = 1.$$

(We use the notation $\iff$ to stand for 'if and only if'.)

So, in fact

$$f(x, y) = \mathrm{sg}(x \dot{-} y).$$

Hence we can get our machine to operate as follows:


$[m] \dot{-} [n] \longrightarrow m$
$0 \longrightarrow n$
$\mathrm{sg}([m]) \longrightarrow p$
$0 \longrightarrow m$

This is achieved by the program of Problem 11 above.

This is achieved by the program of Problem 10 with 'n' replaced by 'p'.

This is achieved by the program of our very first example, Example 1, with 'n' replaced by 'm'.

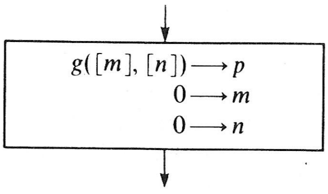
Thus the desired program is obtained by linking together three programs which we have already seen. □

It should be noted that while linking together programs that have already been devised usually provides the easiest way of constructing more complicated programs, it often does not lead to the simplest (measured by the number of instructions) or most efficient (measured by the number of steps to carry out a given computation) program which does the desired job.

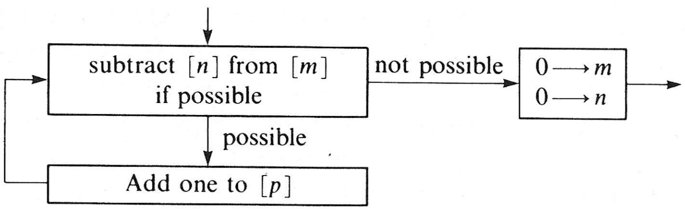
EXAMPLE 5 Our final example deals with the function g defined by

$$g(x, y) = \text{the integer part of } \frac{x}{y},$$

so that, for example, $g(7, 3) =$ the integer part of $2\frac{1}{3} = 2$. We show how to devise a program which has the effect shown in the block diagram.

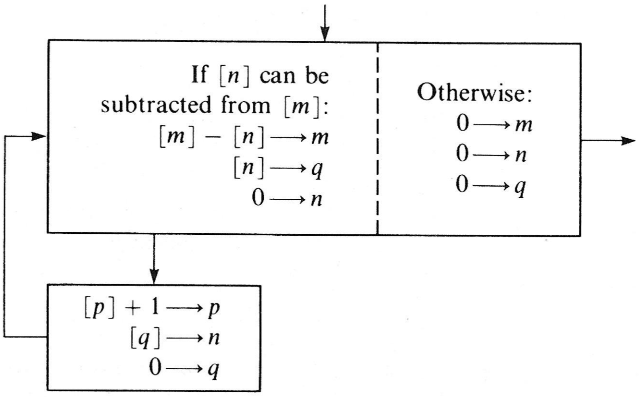


One way to achieve this is to use the fact that $g(x, y)$ equals the number of times y can be subtracted from x without the answer becoming negative. So we are aiming towards a program which will operate as follows:



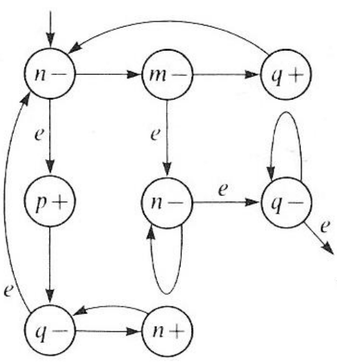
Thus when the computation halts, $[p]$ will equal the number of times we have subtracted $[n]$ from $[m]$, i.e. $g([m], [n])$, provided, of course, that initially $[p] = 0$.

We can use the method of Problem 11 to do the subtraction, but we have to modify its program. That program not only subtracts $[n]$ from $[m]$, but also makes $[n]$ equal to 0, whereas we need to save the original value of $[n]$ for the next subtraction. We met a similar difficulty with the program for exponentiation and we get over it in the same way. As we empty the nth register in the course of subtracting $[n]$ from $[m]$, we fill up register q. When each subtraction is finished, $[q]$ will equal the original value of $[n]$ and we transfer this value back to register n before trying the next subtraction. This leads to the following structure for the program:



The program will discover that it is not possible to subtract $[n]$ from $[m]$ if in the course of trying to do the subtraction it finds that $[m] = 0$ while $[n] \neq 0$.

The program which does all this is as follows:


n−
m−
q+
e
e
p+
n−
e
q−
e
e
q−
n+

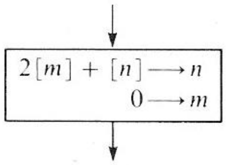
PROBLEM 15 Give the register tables for the computation of the program of Example 5 above in each of the following cases.

(i) Initially $[m] = 7$, $[n] = 3$, $[p] = [q] = 0$.

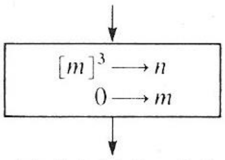
(ii) Initially $[m] = 7$, $[n] = [p] = [q] = 0$.

PROBLEM 16 Devise abacus machines which operate as shown in the following block diagrams. The programs can be given by block diagrams showing how programs that we have already constructed can be fitted together to achieve the desired effect. These exercises are roughly in order of increasing difficulty. We hope that you will try at least the first two or three, but if you are short of time you may prefer to omit the last one or two. (Part (V) is certainly harder than we expect any assessment question to be.)

(i)

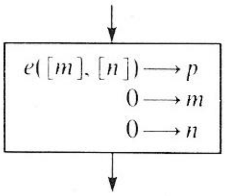

$2[m] + [n] \longrightarrow n$
$0 \longrightarrow m$

(ii)


$[m]^3 \longrightarrow n$
$0 \longrightarrow m$

provided, initially, $[n] = 0$.

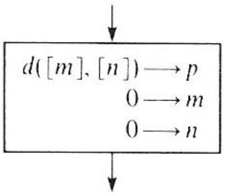
(iii)


$e([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$

where

$$e(x, y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases}$$

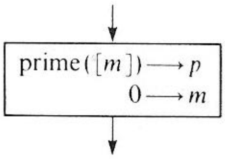
(iv)


$d([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$

where

$$d(x, y) = \begin{cases} 1 & \text{if } x \text{ is divisible by } y, \\ 0 & \text{otherwise.} \end{cases}$$

(v)


prime$([m]) \longrightarrow p$
$0 \longrightarrow m$

where

$$\text{prime}(x) = \begin{cases} 1 & \text{if } x \text{ is prime,} \\ 0 & \text{otherwise.} \end{cases}$$

SOLUTION 15

(i)

[m]	[n]	[p]	[q]
~~7~~	~~3~~	~~0~~	~~0~~
~~6~~	~~2~~	~~1~~	~~1~~
~~5~~	~~1~~	2	~~2~~
~~4~~	~~0~~		~~3~~
~~3~~	~~1~~		~~2~~
~~2~~	~~2~~		~~1~~
~~1~~	~~3~~		~~0~~
0	~~2~~		~~1~~
	~~1~~		~~2~~
	~~0~~		~~3~~
	~~1~~		~~2~~
	~~2~~		~~1~~
	~~3~~		~~0~~
	~~2~~		~~1~~
	~~1~~		0
	0		

(ii)

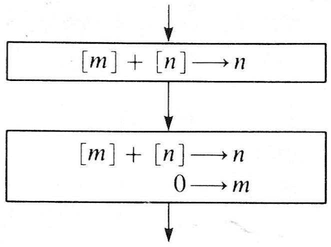
[m]	[n]	[p]	[q]
7	0	~~0~~	0
		~~1~~	
		~~2~~	
		~~3~~	
		4	

In this case the computation does not halt. The machine cycles for ever round the loop made up of the instructions '$n-$', '$p+$' and '$q-$'. Each time it goes round loop the value of $[p]$ is increased by 1. ■

SOLUTION 16

We emphasize again that there are many possible answers to these problems. Because we have made as much use as possible of the programs we have already at our disposal, our solutions may well be more complicated than yours, but we hope that they indicate one way of tackling the problems.

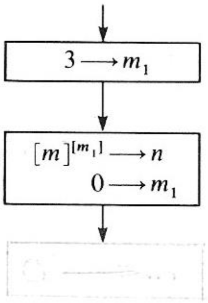
(i)



This is the program of **BJ**, Example 6.3, page 57.

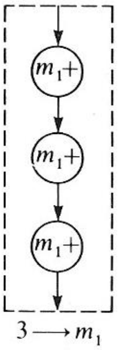
This is the program of **BJ**, Example 6.2, page 56.

(ii) We modify the program of Example 3 in the obvious way.


$3 \longrightarrow m_1$
$[m]^{[m_1]} \longrightarrow n$
$0 \longrightarrow m_1$

Assuming that initially $[m_1] = 0$ this is achieved by the program shown on the right.

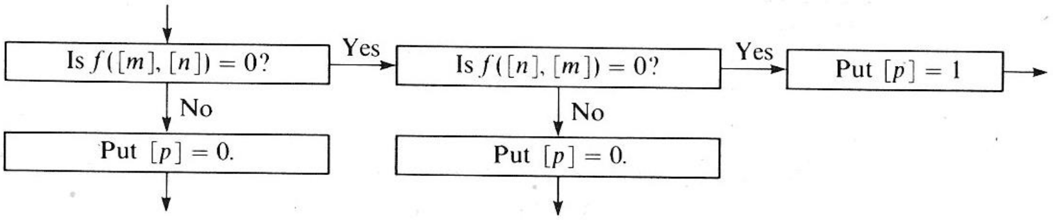
This is the program of Example 6.5 with 'm_2' replaced by 'm'.


m_1+
m_1+
m_1+
$3 \longrightarrow m_1$

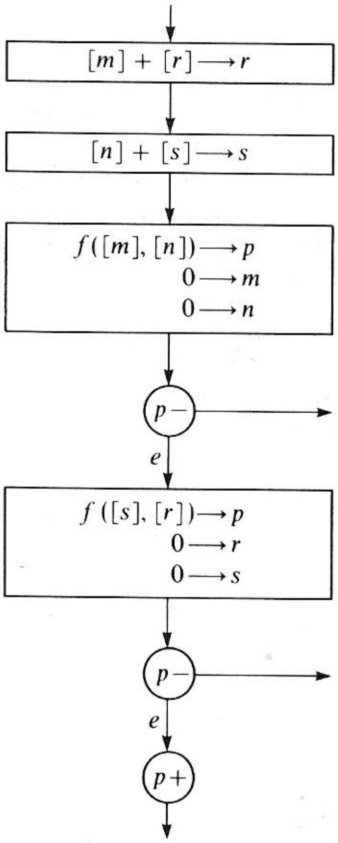
(iii) We make use of the fact that

$$x = y \iff x \le y \text{ and } y \le x$$

and that the function f of Example 4 above can be used to compare the size of two numbers. So we aim to devise a program which does the following. (Note that $x \le y \iff f(x, y) = 0$, so we answer the question 'is $x \le y$?' by answering the question 'is $f(x, y) = 0$?'.)


Is $f([m], [n]) = 0$?
Yes
Is $f([n], [m]) = 0$?
Yes
Put $[p] = 1$
No
Put $[p] = 0$.
No
Put $[p] = 0$.

We run into the familiar technical difficulty that the program of Example 4 for computing $f([m], [n])$ also erases the contents of registers m and n, so we must first copy the original contents of these registers into registers r and s, so they are available for the calculation of $f([n], [m])$. Thus we get the following program.


$[m] + [r] \longrightarrow r$
$[n] + [s] \longrightarrow s$
$f([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$
$p-$
e
$f([s], [r]) \longrightarrow p$
$0 \longrightarrow r$
$0 \longrightarrow s$
$p-$
e
$p+$

This is the program of Example 6.3 with 'n' replaced by 'r'. Provided that initially $[r] = 0$ it has the effect of copying the contents of register m into register r without loss.

Likewise this copies the content of register n without loss into register s.

This is the program of Example 4 above.

If $[p] = 1$ the computation halts with $[p] = 0$ otherwise it continues.

This is the program of Example 4 with 'm', 'n' replaced by 's', 'r'.

If $[p] = 1$ the computation halts with $[p] = 0$,

otherwise it halts with $[p] = 1$.

(iv) Remember that in Example 5 we gave a program for calculating the values of g, where $g(x, y) =$ the integer part of $\frac{x}{y}$. Since

$$x \text{ is divisible by } y \iff g(x, y) \times y = x,$$

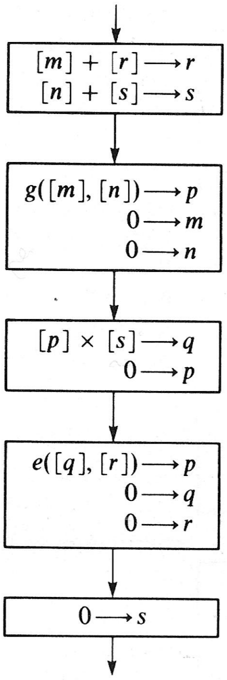
all we need do is carry out the following steps:

(i) calculate $g([m], [n])$,

(ii) multiply the result by $[n]$,

and (iii) check if the answer equals $[m]$.

We already have programs to carry out each of these steps, so all that there remains for us to do is to link them together in a suitable way (once again taking care to make copies of the contents of registers needed later on, that otherwise would get lost). Thus we get:


$[m] + [r] \longrightarrow r$
$[n] + [s] \longrightarrow s$
$g([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$
$[p] \times [s] \longrightarrow q$
$0 \longrightarrow p$
$e([q], [r]) \longrightarrow p$
$0 \longrightarrow q$
$0 \longrightarrow r$
$0 \longrightarrow s$

Transfer the contents of registers m, n without loss to registers r, s where initially $[r] = [s] = 0$

The program of Example 5 above.

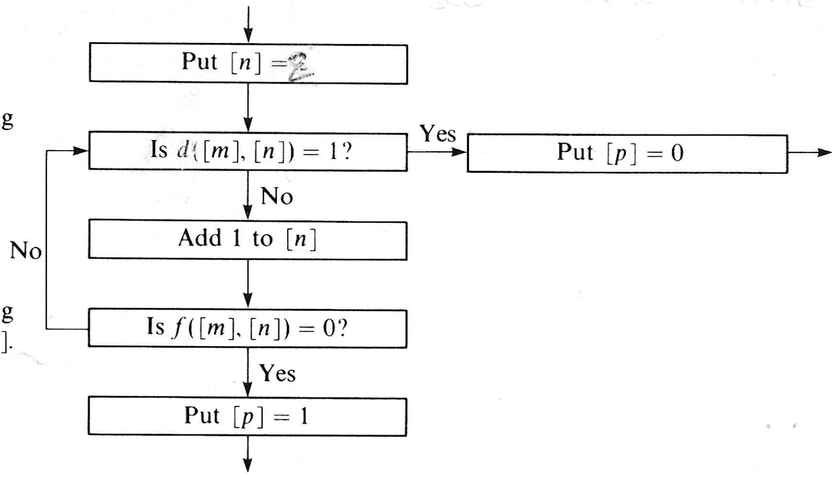
The program of Example 6.4 with the register numbers suitably changed, where initially $[q] = 0$

The program of (iii) above.

This restores $[s]$ to its original value.

(v) To test whether or not x is prime, we use the program of (iv) above to test whether or not x is divisible by 2, 3, 4, …, $x - 1$. That is, we compute successively the values $d(x, 2)$, $d(x, 3)$, …, $d(x, x - 1)$. If we get the value 0, then x is prime; otherwise x is composite. (Of course, to check whether x is prime, it is really necessary only to test x for divisibility by natural numbers less than or equal to $\sqrt{x}$, but to add this refinement would complicate the program and so we omit it.)

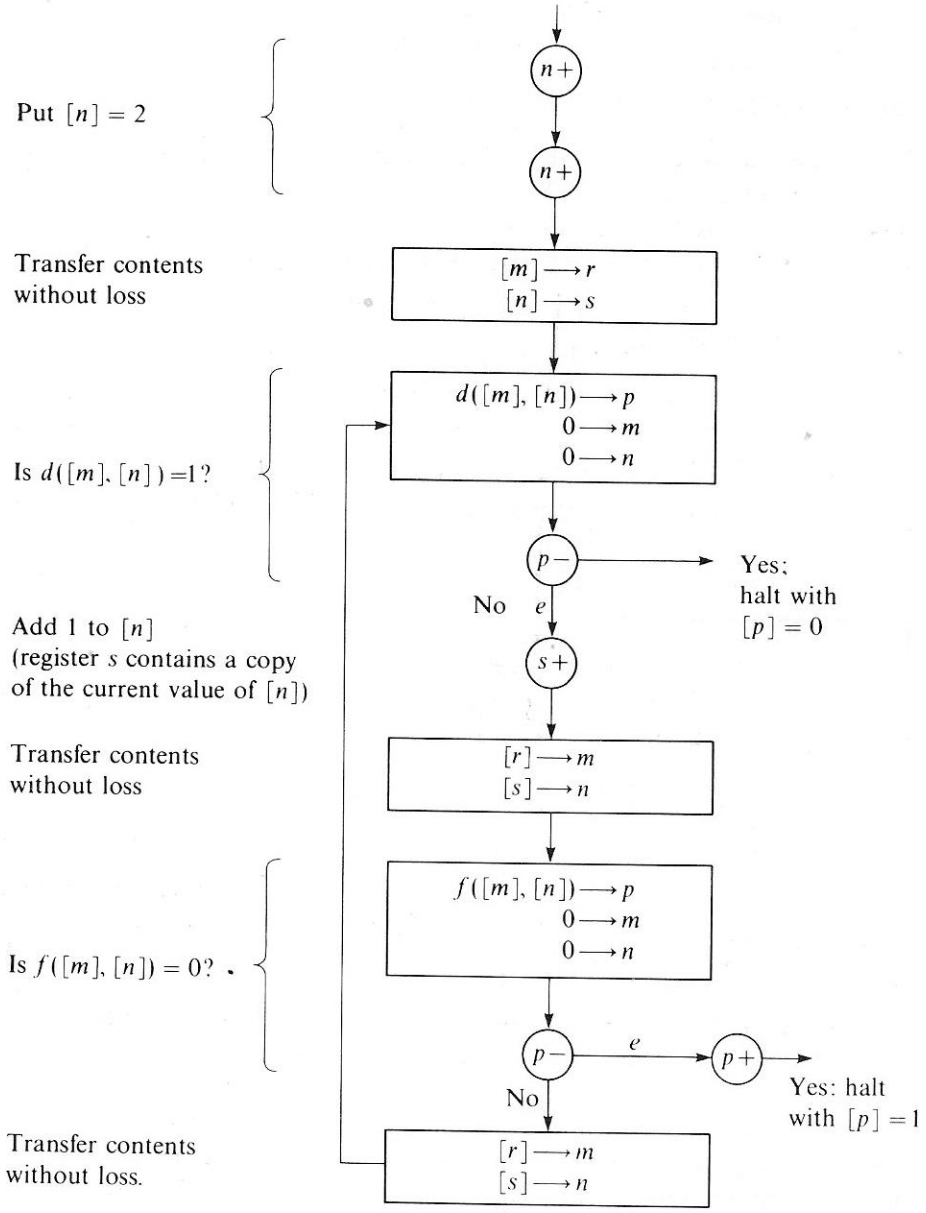
Thus our program will operate as follows:


Put $[n] = 2$
Is $d([m], [n]) = 1$?
Yes
Put $[p] = 0$
No
Add 1 to $[n]$
No
Is $f([m], [n]) = 0$?
Yes
Put $[p] = 1$

Here we are testing whether $[m]$ is divisible by $[n]$.

Here we are testing whether $[m] \leq [n]$.

This is achieved by the following program. The block diagrams which make it up all correspond to previously given programs (usually with a suitable change in the register numbers). As with previous examples, there is the technical problem of making copies of the contents of certain registers to make sure they are not lost in the subsequent computation.


Put $[n] = 2$
$n+$
$n+$
Transfer contents without loss
$[m] \longrightarrow r$
$[n] \longrightarrow s$
Is $d([m], [n]) = 1$?
$d([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$
$p-$
Yes; halt with $[p] = 0$
No
e
Add 1 to $[n]$ (register s contains a copy of the current value of $[n]$)
$s+$
Transfer contents without loss
$[r] \longrightarrow m$
$[s] \longrightarrow n$
Is $f([m], [n]) = 0$?
$f([m], [n]) \longrightarrow p$
$0 \longrightarrow m$
$0 \longrightarrow n$
$p-$
e
$p+$
Yes: halt with $[p] = 1$
No
Transfer contents without loss.
$[r] \longrightarrow m$
$[s] \longrightarrow n$

■

2.3 Abacus Computable Functions are Turing Computable

We hope that the examples and problems which ended the last section will have convinced you that abacus machines are very powerful and can carry out very complicated calculations. Indeed, except for the fact that we have restricted our attention to calculations which involve natural numbers only, we hope that it is now plausible that abacus machines are as powerful as modern digital computers.

In this final section of the unit, we are going to present the proof that, despite appearances to the contrary, Turing machines are no less powerful than abacus machines. We do this by proving that every abacus computable function is also Turing computable. We do not expect you to remember all the technical details of the proof, but we do hope that you will appreciate the main ideas used in it, and that you will remember the strategy of the proof. The list of objectives at the end of the unit will tell you which aspects of the proof will be assessed.

We need first to be clear exactly what it is we are claiming and what we need to do to prove that our claim is true. We said that, for example, the abacus machine of Example 6.4, **BJ** page 58, does multiplication because if it is started with x_1, x_2 as the contents of certain specified registers, then it halts with $x_1 \times x_2$ as the content of another specified register. We can generalize this as follows.

Let f be a function of r arguments. We say that the abacus machine A *computes* f if whenever A is started with $x_1, \ldots, x_r$ as the contents of certain specified registers, then it halts with $f(x_1, \ldots, x_r)$ as the content of another specified register (or if $f(x_1, \ldots, x_r)$ is not defined, it does not halt). We say that f is *abacus computable* if there is some abacus machine A which computes f (and, for technical convenience, we add the requirement that A should use registers $1, \ldots, r$ to store the initial values of the arguments $x_1, \ldots, x_r$).

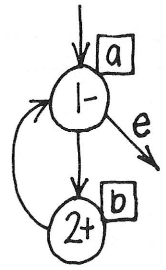
We are claiming that if a function f is abacus computable, then it is also Turing computable. We establish the truth of this claim by giving a general method for converting the flow chart of the abacus machine, A, which computes f, into the flow graph of a Turing machine, T, which imitates the behaviour of A so closely that T also computes the function f.

We illustrate this method by an example on the cassette tape which accompanies this section. You will need to have available pencil and paper so that you can do several calculations as you work through the cassette material.

START THE TAPE WHEN YOU ARE READY

1
2

1



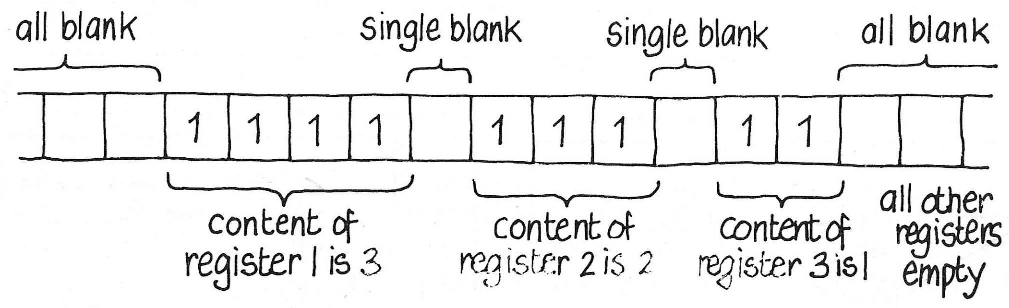
initial contents [1]=3, [2]=2, [3]=1.

2

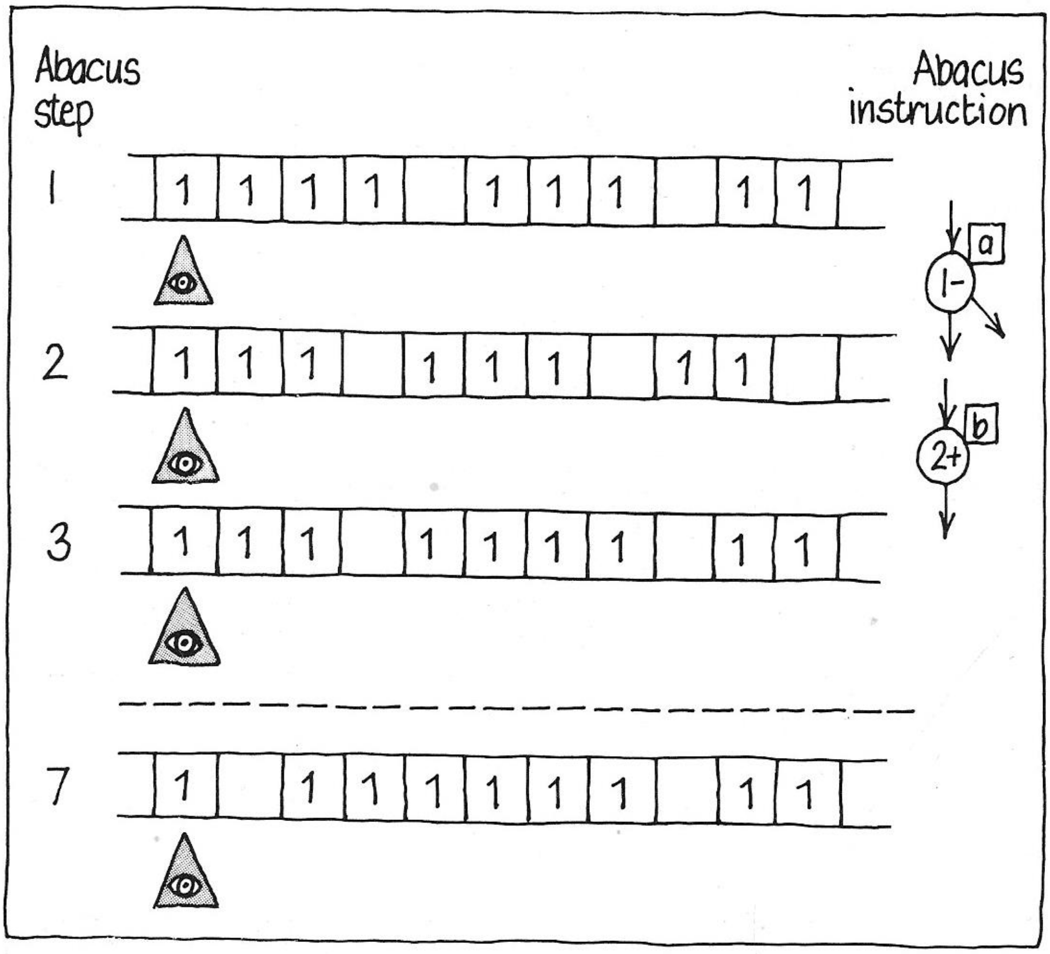
step number	instruction to be performed	content of register [1]	[2]	[3]
1	a	3	2	1
2	b	2	2	1
3	a	2	3	1
4	b	1	3	1
5	a	1	4	1
6	b	0	4	1
7	a	0	5	1

and the computation halts

3


all blank
single blank
single blank
all blank
1 1 1 1
1 1 1
1 1
content of register 1 is 3
content of register 2 is 2
content of register 3 is 1
all other registers empty

4


Abacus step
Abacus instruction
1
1 1 1 1 1 1 1 1 1
2
1 1 1 1 1 1 1 1
3
1 1 1 1 1 1 1 1 1
7
1 1 1 1 1 1 1 1 1
a
1-
b
2+

5

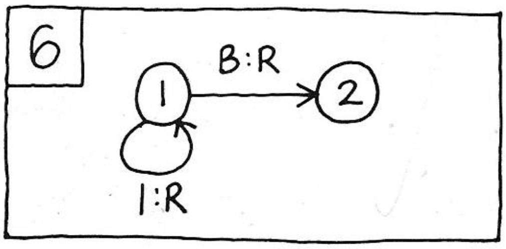
(1) Find the second block of 1s from the left.

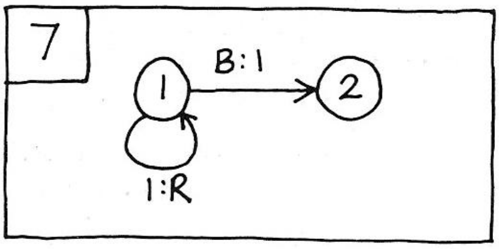
(2) Add a 1 to the right-hand end of this block.

(3) Move each block of 1s which is to the right of the second block one square to the right.

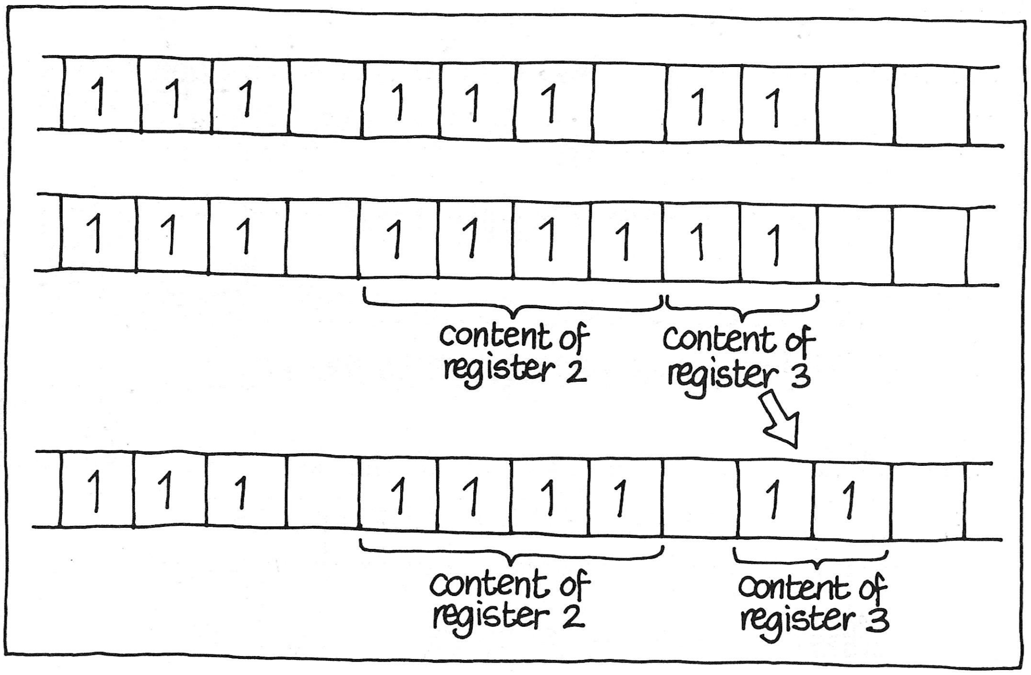
(4) Return the scanning head back to the left-most 1 on the tape.

6
7

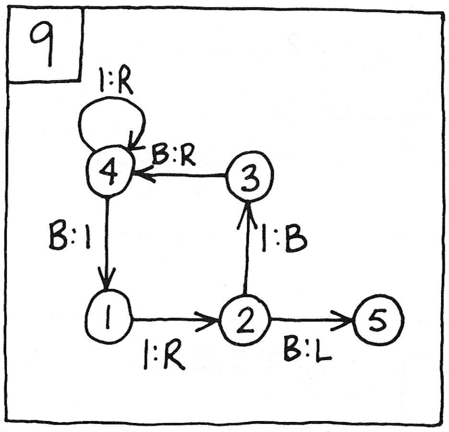


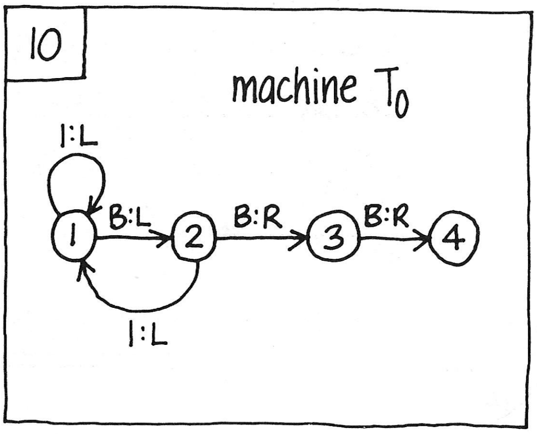


8

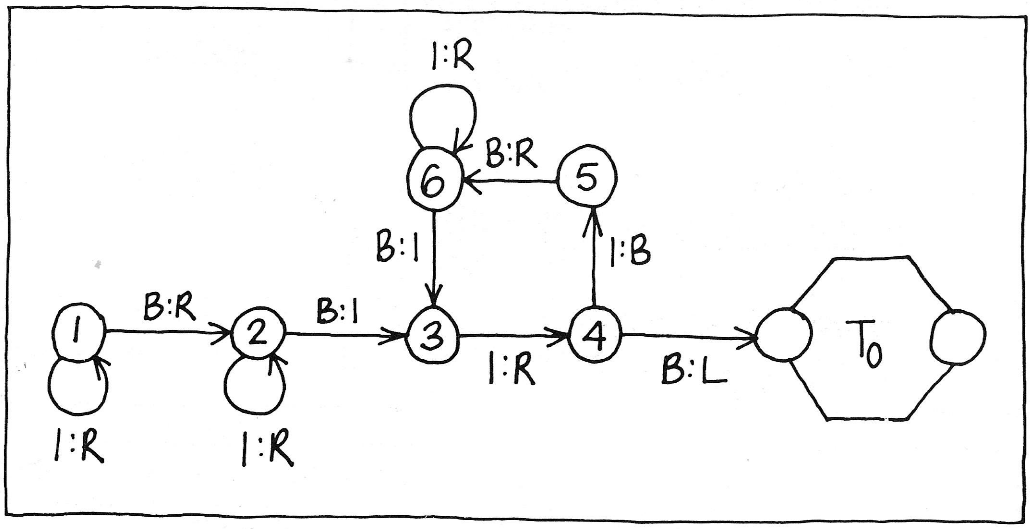

1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
content of register 2
content of register 3
1 1 1 1 1 1 1 1 1
content of register 2
content of register 3

9
10

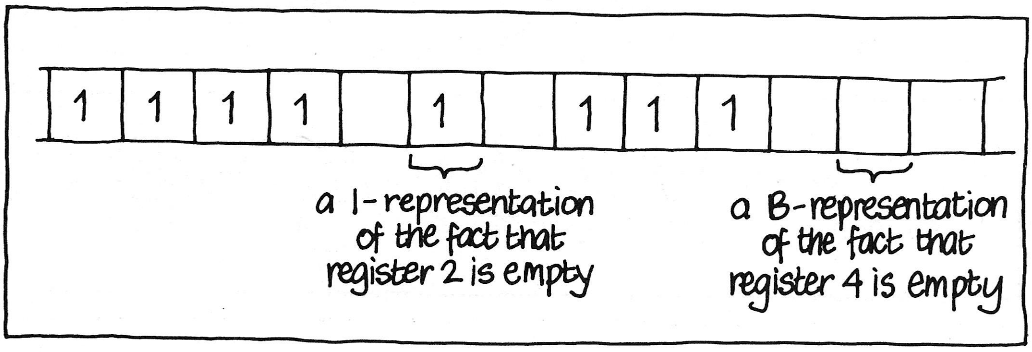

9
1:R
B:R
4
3
B:1
1:B
1
2
5
1:R
B:L


10
machine T0
1:L
B:L
B:R
B:R
1
2
3
4
1:L

11


1:R
B:R
6
5
B:1
1:B
B:R
B:1
1
2
3
4
1:R
B:L
T0
1:R
1:R

12


1 1 1 1 1 1 1 1
a 1-representation of the fact that register 2 is empty
a B-representation of the fact that register 4 is empty

13

(1) Test whether the first block of 1s on the tape represents 0. If so go direct to step (5).

(2) Remove a 1 from the first block.

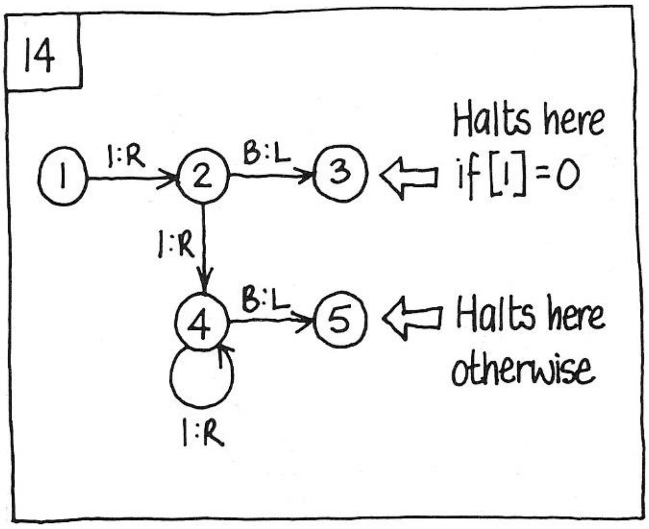
(3) Move all the blocks of 1s which are to the right of the first block one square to the left.

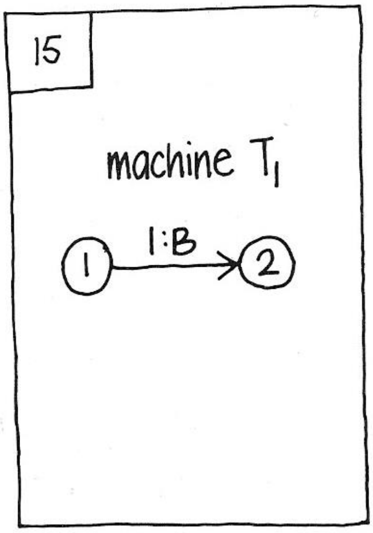
(4) Return the scanning head to the left-most 1 on the tape ready to carry out the relevant instruction for when the first block does not represent zero (i.e. register 1 is not empty).

(5) Return the scanning head to the left-most 1 on the tape ready to carry out the relevant instruction for when the first block does represent zero (i.e. register 1 is empty).

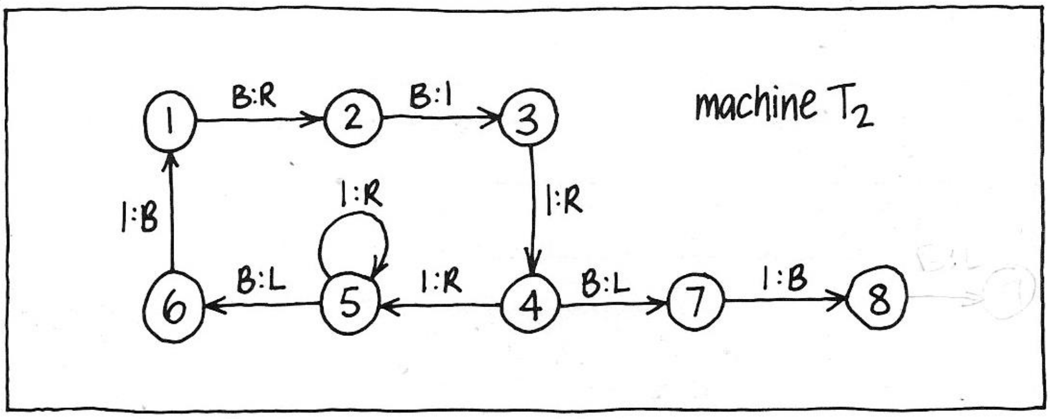
14

15

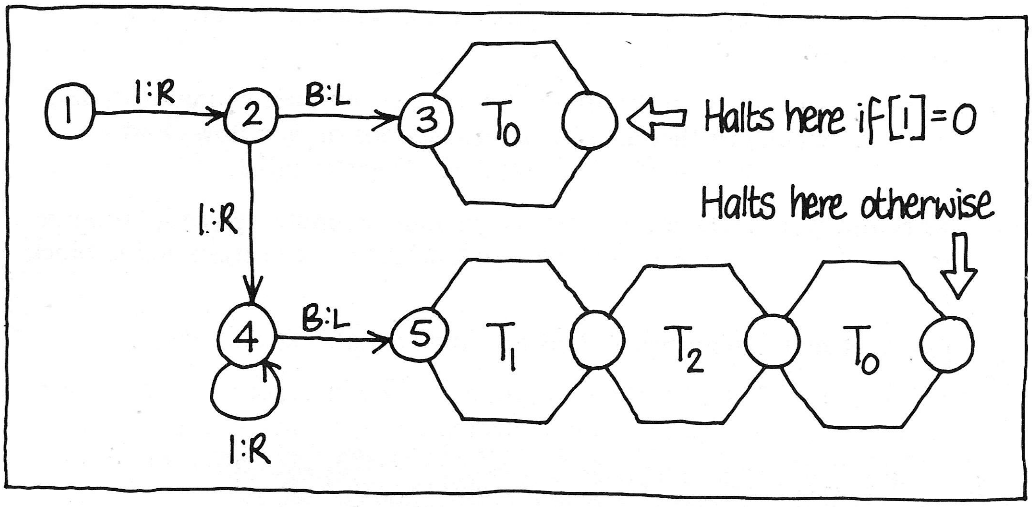

14
1
1:R
2
B:L
3
Halts here if [1] = 0
1:R
4
B:L
5
Halts here otherwise
1:R


15
machine T_1
1
1:B
2

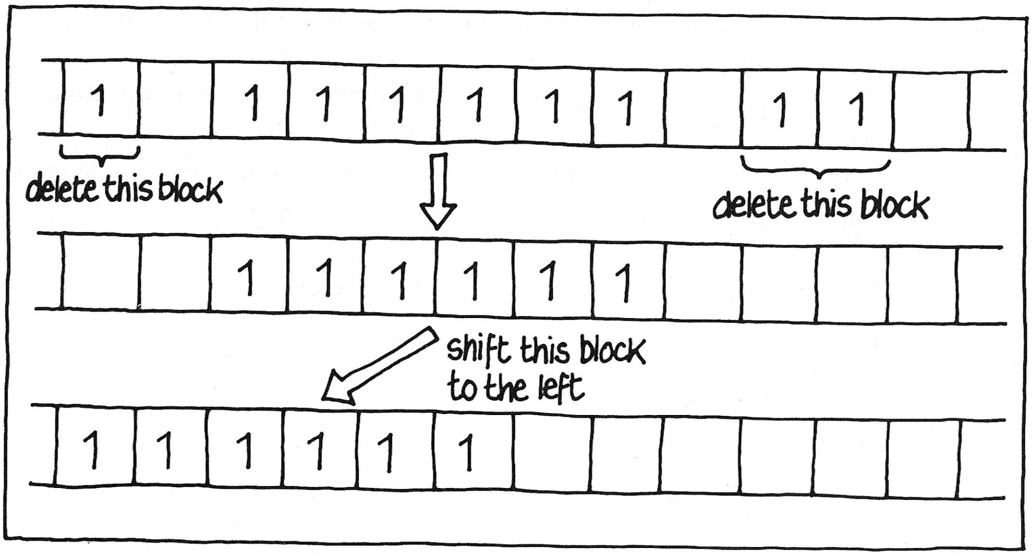
16


machine T_2
1
B:R
2
B:1
3
1:R
1:B
1:R
6
B:L
5
1:R
4
B:L
7
1:B
8

17


1
1:R
2
B:L
3
T_0
Halts here if [1] = 0
1:R
Halts here otherwise
4
B:L
5
T_1
T_2
T_0
1:R

18


delete this block
delete this block
shift this block to the left

BJ discusses the general method for converting an abacus machine into a Turing machine in the reading passage which follows. We have labelled this as 'optional' because everything we hope you will learn from this section is described essentially on the cassette tape. You may, however, find it interesting to read **BJ**'s discussion.

OPTIONAL: **READ BJ** page 60, line 13 to the bottom of page 67 (leaving out Exercises 6.3 and 6.4 on page 66).

We have not provided any notes on this reading passage, because it is optional.

2.4 Objectives of Unit 2

We list those things on which we may set assessment questions to test your understanding of the unit. After working through the unit, we feel you should be able to:

(i) calculate the productivity of a simple Turing machine;

(ii) explain in outline why the fact that the Busy Beaver function *p* is not Turing computable implies that the halting problem is unsolvable by Turing machines;

(iii) convert an abacus program given as a list of instructions into a flow chart, and vice versa;

(iv) work out the behaviour of an abacus machine for specific initial values, given its program either as a list of instructions or as a flow chart or block diagram, by use of a trace table or a register table;

(v) work out which function an abacus machine computes, given its program in any of the ways listed in (iv) above, and draw the corresponding block diagram;

(vi) devise abacus machine programs for doing simple calculations;

(vii) link together abacus machine programs to obtain abacus machines which compute more complicated functions (as a rough guide of the level of difficulty see Problem 16 parts (i), (ii) and (iii) of Section 2.2, rather than part (v) of this problem, which is more difficult);

(viii) outline the proof that shows how a simple abacus computable function is Turing computable, in particular:

  (a) the representation of abacus machine registers on a Turing machine tape;

  (b) the Turing machine tape manipulations corresponding to a single step of an abacus machine.

UNIT 3

RECURSIVE FUNCTIONS

3.0 Introduction

In *Units 1* and *2* we introduced the notions of Turing machine and abacus machine so we could give precise characterizations of what is meant by a computable function. The operations of a Turing machine, you will recall, are extremely basic (especially since we placed the restrictions on them described on **BJ** page 52). We wanted this so as to be as sure as possible that the Turing computable functions are truly mechanically computable and that no human ingenuity or intuition is required in their evaluation. Abacus machines, on the other hand, were introduced for more practical reasons, namely to make the writing of programs easier and to exhibit a theoretical machine closer to a real computer. However, nothing is lost in the definition of abacus computable functions, as we showed by proving that all such functions are, in fact, Turing computable.

We now turn to a more detailed investigation of computable functions. Unfortunately, while our characterizaton in terms of idealized machines is clearly a good way to answer the philosophical question 'which functions are computable?', it does suffer the disadvantage of being rather difficult to handle mathematically.

Consider, for example, the function f of two arguments defined by

$$f(m,n) = (m^2 + 2mn)(m^3 + 5n^4).$$

Clearly this function is computable in the intuitive sense of the word—given particular numbers m and n, familiar algorithms learned at school would suffice to find $f(m,n)$—but how could we show this precisely by proving, for example, that it is Turing computable? The basic method would be to construct the machine table (or flow graph) of a Turing machine which computed its values. This would be a fairly difficult task and it would not be at all clear from the machine table that the function computed by the corresponding Turing machine was indeed f. Constructing an abacus machine program to compute f would also be moderately complicated. Thus the method of constructing a machine for showing that f is computable is far removed from our intuitive understanding that f is computable.

We would suggest that this intuition is in fact based on the realization that f is built up from simple functions using just additions and multiplications (which we know to be Turing computable), and that building up functions in this way will always produce computable functions. Thus the natural way to prove that f is computable is to prove general theorems of the following kind:

(1) if f_1, f_2 are computable so also is $f_1 + f_2$;

(2) if f_1, f_2 are computable so also is $f_1 \times f_2$.

(If f_1, f_2 are both functions of one argument with domain the natural numbers, then by $f_1 + f_2$ we mean, of course, the function with rule $x \longmapsto f_1(x) + f_2(x)$ and domain the natural numbers; similarly for $f_1 \times f_2$. Clearly this definition extends to functions of more arguments.)

We usually express results of this kind by saying that the class of all computable functions is *closed* under the operations of addition and multiplication applied to functions. In general, then, we are led to prove theorems to the effect that the class of computable functions is closed under certain operations.

This suggests an alternative way of giving a precise characterization of the class of computable functions and it is this that we investigate in this unit. We shall define a new class of functions, called the class of *recursive* functions. A function will be called recursive if it can be obtained from a certain small class of *basic* functions (to be specified precisely later) by applying certain operations (also to be specified later). We shall then show that every recursive function is indeed computable—more precisely, abacus computable, which, by the result proved in *Unit 2*, shows that every recursive function is Turing computable.

The discussion above suggests that among our operations should be included addition and multiplication of functions. However, we shall see from the first reading passage that if we examine how general addition can be defined by repeated addition of 1, and general multiplication in terms of repeated additions, it emerges that it is more sensible to use certain other more basic operations for combining functions.

Throughout Sections 3.0 to 3.2 we discuss *total* functions with domain N (the set of natural numbers $\{0, 1, 2, \ldots\}$), i.e. functions defined for all $n \in N$.

READ BJ page 70, line 1 to page 73, line -10.

NOTES (i) *Page 70, lines -2, -1*
Notice that this definition implies that we take $0^0 = 1$; remember we are dealing with functions with domain N, so we have to give some value to 0^0.

(ii) *Page 71, line -9*
The *successor* function s is defined by $s(n) = n + 1$ for $n \in N$.

(iii) *Page 72, example*
The ‘first’ and ‘second’ equations referred to in this example are those on line -3 of page 71. These equations (which, of course, hold for all natural numbers x and y) are to be thought of as a recipe for evaluating $x + y$ given a particular pair of numbers x, y. The second equation tells us how to successively reduce y, and the first tells us what to do when we reach 0.

(iv) *Page 72, line 2*
The phrase ‘computation of’ here (and later) means ‘application of the equations to’ (and not Turing or abacus computation).

(v) *Page 72, line 11*
‘Combining’ here means successively substituting the right-hand side of each of the previous equations into the right-hand side of the equation immediately above it. (Notice that each expression on the right-hand side of one of these equations is of the form $s(E)$ where E is the expression on the left-hand side of the equation immediately below.) ▲

PROBLEM Write out in full the computation (in the current sense of the word) of $2 + 4$.

SOLUTION

$$\begin{aligned}
2 + 4 &= ss0 + ssss0 && \text{(by Definition page 71, line } -8\text{)}\\
&= s(ss0 + sss0) && \text{(by the second equation, page 72, line } -6\text{)}\\
&= ss(ss0 + ss0) && \text{(by the second equation, page 72, line } -6\text{)}\\
&= sss(ss0 + s0) && \text{(by the second equation, page 72, line } -6\text{)}\\
&= ssss(ss0 + 0) && \text{(by the second equation, page 72, line } -6\text{)}\\
&= ssss(ss0) && \text{(by the first equation, page 72, line } -6\text{)}\\
&= ssssss0 &&\\
&= 6 && \text{(by Definition page 71, line } -8\text{)}
\end{aligned}$$

■

3.1 Primitive Recursive Functions

In this section we begin our description of the class of recursive functions. As suggested in Section 3.0, this will be the first step towards the characterization of computable functions in terms of an initial stock of functions and operations for constructing new functions. There will be three such operations and the reading passages of this section specify the class of *basic* functions and two of the operations. The functions that can be constructed using just these two operations are called the *primitive recursive* functions, after one of the operations used. It will become clear that all primitive recursive functions are total functions with domain N.

READ BJ page 73, line −9 to page 76, line 2.

NOTE *Page 75, line 12*
Thus $h(x_1, x_2, x_3) = s(\mathrm{id}_3^3(x_1, x_2, x_3))$, and since id_3^3 extracts the third argument x_3 from (x_1, x_2, x_3), we obtain $h(x_1, x_2, x_3) = s(x_3)$ (= 'the successor of the third argument of h', as required). ▲

Here are some further examples of compositions of functions.

EXAMPLE 1 Consider the function $\mathrm{Cn}[\mathrm{id}_1^2, s, z]$. Since s and z are functions of one argument, this function is also. For $x \in N$ we have

$$\mathrm{Cn}[\mathrm{id}_1^2, s, z](x) = \mathrm{id}_1^2(s(x), z(x)) = \mathrm{id}_1^2(x+1, 0) = x + 1.$$

Thus $\mathrm{Cn}[\mathrm{id}_1^2, s, z]$ is just the successor function, s.

EXAMPLE 2 Let us evaluate $\mathrm{Cn}[\mathrm{Cn}[\mathrm{id}_1^2, s, z], \mathrm{Cn}[s, s]](8)$. From Example 1, we have $\mathrm{Cn}[\mathrm{id}_1^2, s, z] = s$. Thus

$$\begin{aligned}\mathrm{Cn}[\mathrm{Cn}[\mathrm{id}_1^2, s, z], \mathrm{Cn}[s, s]](8) &= \mathrm{Cn}[s, \mathrm{Cn}[s, s]](8)\\ &= s(\mathrm{Cn}[s, s](8))\\ &= s(s(s(8))) = 11.\end{aligned}$$

EXAMPLE 3 We shall show that the function h, defined by $h(x_1, x_2, x_3) = x_2 + 1$ for $x_1, x_2, x_3 \in N$, can be obtained from the basic functions by composition. Since $\mathrm{id}_2^3(x_1, x_2, x_3) = x_2$, for $x_1, x_2, x_3 \in N$, and $s(x_2) = x_2 + 1$, for $x_2 \in N$, we have $h(x_1, x_2, x_3) = s(\mathrm{id}_2^3(x_1, x_2, x_3))$ for $x_1, x_2, x_3 \in N$. Hence $h = \mathrm{Cn}[s, \mathrm{id}_2^3]$.

EXAMPLE 4 We show that the function h, defined by $h(x_1, x_2, x_3) = 2$ for $x_1, x_2, x_3 \in N$, is primitive recursive. Since $z(x_1) = 0$, we have $s(s(z(x_1))) = 2$, for all $x_1 \in N$. To obtain a function of the three arguments (x_1, x_2, x_3) which always takes the value 2, we make use of the function id_1^3. Since $\mathrm{id}_1^3(x_1, x_2, x_3) = x_1$, we get $s(s(z(\mathrm{id}_1^3(x_1, x_2, x_3)))) = 2$ for all $x_1, x_2, x_3 \in N$. Thus $h = \mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[z, \mathrm{id}_1^3]]]$. This equality shows that h may be built up from basic functions using the operation of composition (several times), and so is primitive recursive. (Note that we could have used either id_2^3 or id_3^3 in place of id_1^3 in the expression for h, since, for example, you may easily verify that

$$\mathrm{Cn}[z, \mathrm{id}_1^3](x_1, x_2, x_3) = \mathrm{Cn}[z, \mathrm{id}_2^3](x_1, x_2, x_3), \quad \text{for all } x_1, x_2, x_3 \in N,$$

so that $\mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[z, \mathrm{id}_1^3]]]$ and $\mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[z, \mathrm{id}_2^3]]]$ are the same function, namely h.)

PROBLEM 1 Evaluate $\mathrm{Cn}[\mathrm{id}_2^2, s, \mathrm{id}_1^1](6)$.

PROBLEM 2 Evaluate $\mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{id}_1^3]](2, 3, 4)$.

PROBLEM 3 Prove that the function h defined by $h(x_1, x_2, x_3, x_4) = x_1 + 2$, for $x_1, x_2, x_3, x_4 \in N$, is primitive recursive.

PROBLEM 4 Prove that the function h defined by $h(x_1, x_2) = 3$, for $x_1, x_2 \in N$, is primitive recursive.

PROBLEM 5 Why does the expression $\mathrm{Cn}[z, s, s]$ not define a function?

SOLUTION 1

$$\begin{aligned}\mathrm{Cn}[\mathrm{id}_2^2, s, \mathrm{id}_1^1](6) &= \mathrm{id}_2^2(s(6), \mathrm{id}_1^1(6))\\ &= \mathrm{id}_2^2(7,6)\\ &= 6 \quad \blacksquare\end{aligned}$$

SOLUTION 2

$$\begin{aligned}\mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{id}_1^3]](2,3,4) &= s(\mathrm{Cn}[s, \mathrm{id}_1^3](2,3,4))\\ &= s(s(\mathrm{id}_1^3(2,3,4)))\\ &= s(s(2))\\ &= s(3)\\ &= 4 \quad \blacksquare\end{aligned}$$

SOLUTION 3 For $x_1, x_2, x_3, x_4 \in N$, we have

$$h(x_1,x_2,x_3,x_4) = x_1 + 2 = s(s(x_1)) = s(s(\mathrm{id}_1^4(x_1,x_2,x_3,x_4))).$$

Hence $h = \mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{id}_1^4]]$, which shows h to be primitive recursive. $\blacksquare$

SOLUTION 4 For $x_1, x_2 \in N$, $h(x_1,x_2) = 3 = s(s(s(0))) = s(s(s(z(x_1)))) = s(s(s(z(\mathrm{id}_1^2(x_1,x_2))))).$

Thus $h = \mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[z, \mathrm{id}_1^2]]]]$, and h is therefore primitive recursive. (As in Example 4 above, id_1^2 may be replaced by id_2^2 in the expression for h here.) $\blacksquare$

SOLUTION 5 In general, if f is a function of k arguments then $\mathrm{Cn}[f, g_1, \ldots, g_m]$ is defined only if $m = k$, i.e. if the number of functions listed after f is the same as the number of arguments of f. Here z is a function of 1 argument and so $\mathrm{Cn}[z, s, s]$ (where $m = 2$) is not defined. $\blacksquare$

The next reading passage introduces the second operation used to construct the recursive functions. It is called *primitive recursion*. We have, in fact, met examples of this in Section 3.0. Consider, for example, the equations defining the exponential function given on **BJ** page 70:

$$\text{for all } x, y \in N, \qquad \text{(i) } x^0 = 1, \quad \text{(ii) } x^{y+1} = x \cdot x^y.$$

If we rewrite these equations using s for the successor function, $\mathrm{prod}(x,y)$ for $x \cdot y$, and $\exp(x,y)$ for x^y, they look like:

$$\text{for all } x, y \in N, \qquad \text{(i) } \exp(x,0) = 1, \quad \text{(ii) } \exp(x, s(y)) = \mathrm{prod}(x, \exp(x,y)).$$

These equations enable us to calculate $\exp(x,y)$ for any given natural numbers x, y (assuming we can calculate $\mathrm{prod}(x,y)$ for any natural numbers x, y) by calculating successively the values

$$\exp(x,0), \exp(x,1), \exp(x,2), \ldots$$

until we eventually reach $\exp(x,y)$. Equation (i) tells us how to *start* this calculation (by setting $\exp(x,0) = 1$), and equation (ii) tells us how to *continue* (by multiplying the previous number by x). This process was illustrated in Section 3.0, where the case $x = 5$, $y = 3$ was computed in full (**BJ** page 71).

The general form of the definition of a function h (of two variables) by primitive recursion will consist of two equations. The first,

$$\text{(i) } h(x,0) = \ldots,$$

tells us how to start the calculation, and the second,

$$\text{(ii) } h(x, s(y)) = \ldots,$$

how to continue it.

In the case of exp, the value of $\exp(x, 0)$ given on the right-hand side of (i) is always 1, but in general we can expect $h(x, 0)$ to vary with x. So, in general, on the right-hand side of equation (i) we shall have some *previously defined* function of x, say $x \longmapsto f(x)$ (for $x \in N$). To see what form the right-hand side of equation (ii) should take, note that in the case of exp, the value of $\exp(x, s(y))$ depends on the previously calculated value of $\exp(x, y)$. Thus $h(x, s(y))$ should depend on the previous value $h(x, y)$. In fact, $\exp(x, s(y))$ is a function of $\exp(x, y)$ *and* x (namely $\text{prod}(x, \exp(x, y))$), and in the general situation we shall allow $h(x, s(y))$ to be a *previously defined* function, say g, of x, y and $h(x, y)$. Thus we arrive at the definition of primitive recursion given by **BJ**. ('Recursion' is a purely mathematical word, and is meant to convey the idea of calculating some value (of a function) from previously calculated values.)

READ BJ page 76, line 3 to page 77, line 10.

NOTE *Page 76, lines 12 to −5*

Here **BJ** shows that the definition of addition already given (see **BJ** page 71, line −3) can be made to fit exactly the format of a definition by primitive recursion as given in the box towards the top of the page. The values of a general function h of two arguments are written as $h(x, y)$, but for addition we usually use the special symbol '+' written between x and y. So the first move is to give the addition function the name 'sum' and to write '$\text{sum}(x, y)$' instead of '$x + y$'. Secondly, we have seen that on the right-hand side of the first equation there should be some function of x. But on the right-hand side of the equation $\text{sum}(x, 0) = x$ we have just x by itself. So we need to replace this by a function which, when applied to x, yields x itself as value, and of course the identity function, id, is just what we need here. Finally, on the right-hand side of the second equation we should have some function of x, y and $\text{sum}(x, y)$, whereas in '$\text{sum}(x, s(y)) = s(\text{sum}(x, y))$' all we have is just a function of $\text{sum}(x, y)$. So we are looking for a function of three arguments, $(x, y, \text{sum}(x, y))$, which has value $s(\text{sum}(x, y))$, and $\text{Cn}[s, \text{id}_3^3]$ does this for us.

Thus the function sum is defined by primitive recursion from the functions id and $\text{Cn}[s, \text{id}_3^3]$. This definition is expressed by the single equation

$$\text{sum} = \text{Pr}[\text{id}, \text{Cn}[s, \text{id}_3^3]]. \quad \blacktriangle$$

You may have observed that the definition of primitive recursion constructs only functions of two arguments. (Thus Pr is an operation which, when applied to a function f of one argument and a function g of three arguments produces a function h of two arguments.) We give a more general form after the following example and problems. For the moment let us recall the definition of a primitive recursive function. It is a function that can be obtained from the initial functions by a finite number of applications of the operations Pr and Cn. Thus the equation

$$\text{sum} = \text{Pr}[\text{id}, \text{Cn}[s, \text{id}_3^3]]$$

tells us immediately that the function sum is primitive recursive.

Also, the equation

$$\text{prod} = \text{Pr}[z, \text{Cn}[\text{sum}, \text{id}_1^3, \text{id}_3^3]]$$

tells us that prod is primitive recursive, since sum is the only non-basic function appearing here, and we have already shown this to be primitive recursive. We could, of course, replace the occurrence of sum here by $\text{Pr}[\text{id}, \text{Cn}[s, \text{id}_3^3]]$ to demonstrate explicitly that prod can be obtained from the initial functions by applications of the operations Cn and Pr. The point is that once a function has been shown to be primitive recursive, we may apply to it the operations Cn and Pr, together with other suitable functions, to obtain further primitive recursive functions.

EXAMPLE 5 We calculate some values of the function $\mathrm{Pr}[s, \mathrm{Cn}[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3]]$. Let us denote this function by h, and the function $\mathrm{Cn}[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3]$ by g. Then $g(x_1, x_2, x_3) = \mathrm{prod}(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3)) = x_1 \cdot x_3$. Thus the equations for h are (for $x, y \in N$)

$$h(x,0) = s(x), \quad h(x, s(y)) = g(x, y, h(x,y)) = x \cdot h(x,y).$$

(These are obtained by substituting s for f and $\mathrm{Cn}[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3]$ for g in the definition of Pr in **BJ** page 76, line 5.) We use these equations to calculate $h(3,3)$ and $h(2,4)$ as follows:

$h(3,0) = s(3) = 4$ (from the first equation with $x = 3$);

$h(3,1) = h(3, s(0)) = 3 \cdot h(3,0) = 3 \cdot 4 = 12$ (from the second equation with $x = 3$, $y = 0$);

$h(3,2) = h(3, s(1)) = 3 \cdot h(3,1) = 3 \cdot 12 = 36$ (similarly with $x = 3$, $y = 1$);

$h(3,3) = h(3, s(2)) = 3 \cdot h(3,2) = 3 \cdot 36 = 108$ (again using the second equation with $x = 3$, $y = 2$).

The calculation of $h(2,4)$ is similar:

$$h(2,0) = s(2) = 3$$

$$h(2,1) = 2 \cdot h(2,0) = 6$$

$$h(2,2) = 2 \cdot 6 = 12$$

$$h(2,3) = 2 \cdot 12 = 24$$

$$h(2,4) = 2 \cdot 24 = 48.$$

PROBLEM 6 Let h be the function $\mathrm{Pr}[\mathrm{Cn}[\mathrm{prod}, \mathrm{id}, s], \mathrm{Cn}[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3]]$. Calculate $h(3,0)$ and $h(2,2)$.

PROBLEM 7 Prove that the function exp is primitive recursive.

SOLUTION 6 We have $\mathrm{Cn}[\mathrm{prod}, \mathrm{id}, s](x) = \mathrm{prod}(\mathrm{id}(x), s(x)) = x \cdot (x+1)$ (for $x \in N$)

$$\text{and } \mathrm{Cn}[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3](x_1, x_2, x_3) = \mathrm{sum}(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3))$$

$$= x_1 + x_3 \qquad \text{(for } x_1, x_2, x_3 \in N\text{)}.$$

Thus the equations for h are

$$h(x,0) = x \cdot (x+1) \quad \text{and} \quad h(x, s(y)) = x + h(x,y).$$

Therefore,

$$h(3,0) = 3 \cdot (3+1) = 12$$

$$h(2,0) = 2 \cdot (2+1) = 6$$

$$h(2,1) = 2 + h(2,0) = 2 + 6 = 8$$

$$h(2,2) = 2 + h(2,1) = 2 + 8 = 10. \quad ■$$

SOLUTION 7 We use the equations $\exp(x,0) = 1$ and $\exp(x, s(y)) = \mathrm{prod}(x, \exp(x,y))$ which were introduced at the beginning of this section. To get them into exactly the correct format we must write '1' as a function of x, and $\mathrm{prod}(x, \exp(x,y))$ as a function of x, y and $\exp(x,y)$. However, $1 = s(z(x))$, for all $x \in N$, and

$$\mathrm{prod}(x, \exp(x,y)) = \mathrm{prod}(\mathrm{id}_1^3(x, y, \exp(x,y)), \mathrm{id}_3^3(x, y, \exp(x,y))).$$

Thus if we take $f = \mathrm{Cn}[s, z]$ and $g = \mathrm{Cn}[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3]$, so that f and g are clearly primitive recursive, we see that

$$\exp(x,0) = f(x), \quad \exp(x, s(y)) = g(x, y, \exp(x,y)),$$

i.e. $\exp = \mathrm{Pr}[f, g]$. Hence exp is primitive recursive. ■

As remarked already, so far we have specified primitive recursion only as a method for defining functions of two arguments. We conclude this section with a short passage in which **BJ** gives the full form of primitive recursion which can be used to define functions of any number of arguments—not just two.

READ BJ page 77, line -12 to page 78, line 2.

From now on, by *primitive recursive function* we shall mean any function that can be built up from the basic functions using the operations of composition and our new general form of primitive recursion.

EXAMPLE 6 The function pred is defined by

$$\text{pred}(y) = \begin{cases} 0, & \text{if } y = 0, \\ y - 1, & \text{if } y > 0. \end{cases}$$

('pred' stands for *predecessor*. Of course 0 has no predecessor in the natural numbers but we must give pred(0) some value in N and 0 is the most convenient.) We show that pred is primitive recursive by observing that it satisfies the equations

$$\text{pred}(0) = 0, \quad \text{pred}(s(y)) = y \text{ for all } y \in N.$$

If we rewrite the second equation as $\text{pred}(s(y)) = \text{id}_1^2(y, \text{pred}(y))$, we see that pred fits **BJ**'s format for defining a function of one argument by primitive recursion (with $h = \text{pred}$ and $g = \text{id}_1^2$—see **BJ** page 77, lines -2, -1), and so pred is primitive recursive.

EXAMPLE 7 In the previous example, we easily wrote down two equations satisfied by pred which were in nearly the correct format for a primitive recursion, but then had to make a trivial (but tiresome) use of an identity function to get these equations into exactly the right format. We now prove a general result which allows us to dispense with this latter operation.

Let f and g be primitive recursive functions of one argument. Suppose the function h (of two arguments) satisfies the equations

$$h(x,0) = f(x), \quad h(x,s(y)) = g(h(x,y)), \quad \text{for } x, y \in N.$$

Then h is primitive recursive: for we can rewrite the second equation as $h(x,s(y)) = \text{Cn}[g, \text{id}_3^3](x,y,h(x,y))$ which, together with the equation $h(x,0) = f(x)$, shows that h fits exactly the format of a primitive recursion (**BJ** page 76, line 5).

EXAMPLE 8 As an application of Example 7, consider the function dif defined by

$$\text{dif}(x,y) = \begin{cases} x - y, & \text{if } x \geq y, \\ 0, & \text{if } x < y. \end{cases}$$

(We met dif, the difference function, in Problem 11 of Section 2.2 of *Unit 2*, where we used the notation $x \dot{-} y$ for $\text{dif}(x,y)$.) You can verify that this function satisfies the equations

$$\text{dif}(x,0) = x, \text{ i.e. } \text{dif}(x,0) = \text{id}(x), \quad \text{dif}(x,s(y)) = \text{pred}(\text{dif}(x,y)),$$

so it is primitive recursive by Example 7 (taking $f = \text{id}$ and $g = \text{pred}$). ☐

In the following problems we ask you to prove that certain functions are primitive recursive. You may, of course, use any functions that have already been shown to be primitive recursive, as well as any short cut that can be justified easily using the method of Example 7 above.

It might be helpful to look at **BJ** page 84 at this point, where there is a list of most of the functions we have shown so far to be primitive recursive. Both the exact definitions and more informal definitions are given there—we suggest you follow the more informal style once you are satisfied that you could, if necessary, convert such definitions to the exact formats.

PROBLEM 8 **BJ** page 78, Exercise 7.3.

PROBLEM 9 (i) Suppose that f is a primitive recursive function of two arguments and n is any fixed natural number. Prove that the function h of one argument defined by

$$h(y) = f(n, y), \quad \text{for } y \in N$$

is primitive recursive.

(ii) Deduce that the functions $x \longmapsto 2x\ (x \in N)$, $x \longmapsto 3x\ (x \in N)$, and $x \longmapsto 5x\ (x \in N)$ are primitive recursive.

PROBLEM 10 The function adf (*absolute difference*) is defined by

$$\operatorname{adf}(x, y) = |x - y| = \begin{cases} x - y, & \text{if } x \geq y, \\ y - x, & \text{if } x < y. \end{cases}$$

Prove that adf is primitive recursive.

PROBLEM 11 Show that if the function f (of one argument, is primitive recursive, then so also is the function g defined by:

$$g(x) = \sum_{n<x} f(n) = f(0) + f(1) + \cdots + f(x - 1),$$

i.e. $g(x)$ equals the sum of the first x values of f, where we make the convention that

$$g(0) = \sum_{n<0} f(n) = 0.$$

PROBLEM 12 Show that if f_1 and f_2 are primitive recursive functions of n arguments, then so also are the functions g and h defined by

$$g(x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) + f_2(x_1, \ldots, x_n),$$
$$h(x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) \cdot f_2(x_1, \ldots, x_n).$$

(Notice that this provides some justification for our claim in Section 3.0 that the operations Cn and Pr are more basic than addition and multiplication. This problem shows that we can deduce closure of the class of primitive recursive functions under the operations of addition and multiplication of functions from closure under Cn and Pr.)

PROBLEM 13 The functions sg and $\overline{\text{sg}}$ are defined as follows

$$\operatorname{sg}(x) = \begin{cases} 0, & \text{if } x = 0, \\ 1, & \text{if } x > 0, \end{cases}$$

$$\overline{\operatorname{sg}}(x) = \begin{cases} 1, & \text{if } x = 0, \\ 0, & \text{if } x > 0. \end{cases}$$

Prove that sg and $\overline{\text{sg}}$ are primitive recursive. (We met sg, the *signum* function, in Problem 10 of Section 2.2 of *Unit 2*.)

PROBLEM 14 Prove that the function e defined by

$$e(x) = \begin{cases} 0, & \text{if } x \text{ is even}, \\ 1, & \text{if } x \text{ is odd}, \end{cases}$$

is primitive recursive. (We met this function in Problem 6 of Section 1.3 of *Unit 1*, where it was called the parity function.)

PROBLEM 15 Define the function h by

$h(x)$ = the number of odd numbers which are less than x.

$$\left(\text{Thus } h(x) = \begin{cases} \dfrac{x}{2}, & \text{if } x \text{ is even,} \\ \dfrac{x-1}{2}, & \text{if } x \text{ is odd.} \end{cases}\right)$$

Prove that h is primitive recursive. [Hint: Use Problems 11 and 14.]

PROBLEM 16 Prove that the function $x \longmapsto 2^x$ $(x \in N)$ is primitive recursive.

PROBLEM 17 In *Unit 4*, we shall use the functions pr ('pair') and tpl ('triple') which are defined by

$$\begin{aligned} \mathrm{pr}(x, y) &= 2^x \cdot 3^y, \\ \mathrm{tpl}(x, y, z) &= 2^x \cdot 3^y \cdot 5^z. \end{aligned}$$

Prove that pr and tpl are primitive recursive.

SOLUTION 8 A solution appears as Example 7.4 on **BJ** page 84, line -13, with the more informal description on line -5, where the more usual notation $y!$ is used for $\mathrm{fac}(y)$. ■

SOLUTION 9 (i) If j is the function of one argument taking constant value n (i.e. $j(x) = n$ for all $x \in N$), then $h = \mathrm{Cn}[f, j, \mathrm{id}]$. Hence it is sufficient to show that j is primitive recursive. But $j = \mathrm{Cn}[s, \mathrm{Cn}[s, \mathrm{Cn}[s, \ldots, \mathrm{Cn}[s, z] \ldots]]]$, where there are n applications of Cn, which shows j is primitive recursive. Hence h is primitive recursive.

(ii) Clearly, for all $x \in N$ we have $2x = \mathrm{prod}(2, x)$, $3x = \mathrm{prod}(3, x)$ and $5x = \mathrm{prod}(5, x)$. So the result follows from part (a), since prod is primitive recursive. ■

SOLUTION 10 A solution appears on **BJ** page 84, line -1. One can easily prove this by considering separately the cases $x \geq y$ and $x < y$. Formally, $\mathrm{adf} = \mathrm{Cn}[\mathrm{sum},$ [illegible]$, \mathrm{Cn}[\mathrm{dif}, \mathrm{id}_2^2, \mathrm{id}_1^2]]$. ■

SOLUTION 11 Notice that for all $x \in N$,

$$\begin{aligned} g(s(x)) &= (f(0) + f(1) + \cdots + f(x-1)) + f(x) \\ &= g(x) + f(x), \end{aligned}$$

so $g(0) = 0$ and

$$\begin{aligned} g(s(x)) &= \mathrm{sum}\,(f(x), g(x)) \\ &= \mathrm{Cn}[\mathrm{sum}, \mathrm{Cn}[f, \mathrm{id}_1^2], \mathrm{id}_2^2]\,(x, g(x)). \end{aligned}$$

As $\mathrm{Cn}[\mathrm{sum}, \mathrm{Cn}[f, \mathrm{id}_1^2], \mathrm{id}_2^2]$ is primitive recursive, so also is g. ■

SOLUTION 12 $g = \mathrm{Cn}[\mathrm{sum}, f_1, f_2]$, $h = \mathrm{Cn}[\mathrm{prod}, f_1, f_2]$. ■

SOLUTION 13 Informal definitions appear in **BJ** page 85, lines 1, 2. Notice the use of Problem 9(i) here, for example with $f = \mathrm{dif}$ and $n = 1$, in the definition of $\overline{\mathrm{sg}}$. ■

SOLUTION 14 Notice that e satisfies the equations

$$e(0) = 0, \quad e(x+1) = \begin{cases} 0, & \text{if } e(x) = 1, \\ 1, & \text{if } e(x) = 0, \end{cases}$$

(since if x is even, then $x + 1$ is odd, and vice versa).

We may rewrite these equations using the function $\overline{\mathrm{sg}}$ of Problem 13 above:

$$e(0) = 0, \quad e(s(x)) = \overline{\mathrm{sg}}\,(e(x)),$$

which shows that e is primitive recursive. (We have, in fact, used a modification of Example 7 above for a function h of only one argument.) ■

SOLUTION 15 Consider the sum $e(0) + e(1) + \cdots + e(x-1)$. In this sum, each odd number between 0 and $x - 1$ contributes a 1, and each even number contributes 0. Thus the value of this sum is exactly the number of odd numbers which are less than x. Thus

$$h(x) = \sum_{n<x} e(n),$$

so h is primitive recursive by Problems 14 and 11 above. ■

SOLUTION 16 We have

$$2^0 = 1, \quad 2^{s(x)} = 2 \cdot 2^x \quad \text{(for all } x \in N),$$

which shows that $x \longmapsto 2^x \; (x \in N)$ is primitive recursive by Problem 9(ii) (first part) and a modification of Example 7 above to one argument functions. Alternatively, note that for all $x \in N$, $2^x = \exp(2, x)$ so the result also follows from Problems 7 and 9(i) above. ■

SOLUTION 17 Here we may use either the primitive recursive equations

$$\mathrm{pr}(x, 0) = 2^x, \quad \mathrm{pr}(x, s(y)) = 3 \cdot \mathrm{pr}(x, y),$$

or first prove that the function $x \longmapsto 3^x \; (x \in N)$ is primitive recursive by a similar method to Problem 16, then note that

$$\mathrm{pr}(x, y) = \mathrm{prod}(2^x, 3^y)$$

and use composition. (Both solutions here require the second part of Problem 9(ii).)

Similarly for tpl:

$$\mathrm{tpl}(x, y, 0) = \mathrm{pr}(x, y), \quad \mathrm{tpl}(x, y, s(z)) = 5 \cdot \mathrm{tpl}(x, y, z),$$

or

$$\mathrm{tpl}(x, y, z) = \mathrm{prod}(2^x, \mathrm{prod}(3^y, 5^z)) \quad \text{(for all } x,\ y,\ z \in N). \; ■$$

3.2 Primitive Recursive Functions are Abacus Computable

In this section we shall show that given any primitive recursive function f, of say n arguments, there is an abacus machine which computes f. That is, we shall construct an abacus program having the following block diagram:

$$\begin{array}{c} \downarrow \\ \boxed{f([1], \ldots, [n]) \longrightarrow n+1} \\ \downarrow \end{array}$$

(We have to specify some register for the result of the computation to be stored in, and the $(n + 1)$th is the most convenient for current purposes.)

To achieve our aim we need to do three things:

(i) prove that all the basic functions are abacus computable;

(ii) prove that if f is obtained by the composition of abacus computable functions then f is abacus computable;

(iii) prove that if f is obtained by primitive recursion from abacus computable functions then f is abacus computable.

Since every primitive recursive function is obtained from the basic functions by a finite number of uses of composition and primitive recursion, it will then follow that every primitive recursive function is abacus computable.

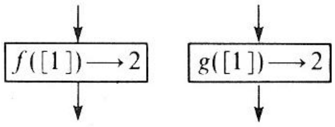
For (ii) above, we must show how to construct an abacus which computes the function $\mathrm{Cn}[f, g_1, \ldots, g_m]$ whenever we are given abaci for computing the functions $f, g_1, \dots, g_m$. In general, f will be a function of m arguments and each g_i will be a function of n arguments. However, in the reading passage below, **BJ** does the construction for the case $n = 3$, $m = 2$ in order to achieve greater clarity (and argues that the general case is completely analogous). If you have trouble understanding this construction, we have described the program for the simplest case ($n = 1$ and $m = 1$) in the example below and suggest you look at that before re-reading **BJ**'s example. **BJ** then goes on to establish (iii) in the case where f is a function of two arguments.

READ BJ page 78, line 7 to page 81, line −6, including the diagram on page 82.

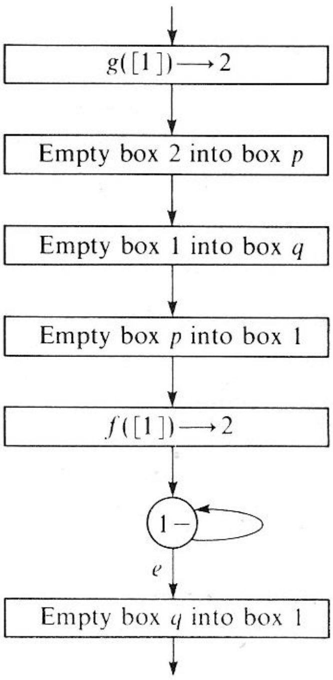
NOTE *Page 78, Figure 7-1*
If the program for the zero function seems strange, think of what we want the abacus to do. Starting with the number x in register 1 and all other registers empty, the computation must terminate with x in register 1 and $z(x)$ in register 2—i.e. 0 in register 2. The easiest way to achieve this is to do nothing!

In the second and third diagrams, $[2]$ and $[n+1]$ respectively are, of course, both zero since we assume registers 2 and $n+1$, respectively, are both empty initially. **BJ** has written the block diagrams in this form so that they correspond exactly to **BJ** Example 6.3. ▲

EXAMPLE Suppose we have abacus programs for computing the functions f and g, each of one argument, with block diagrams

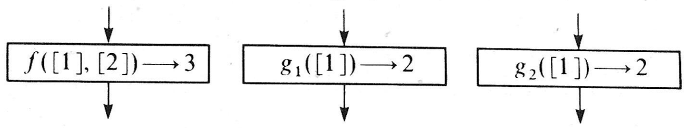


On the left below, we give the block diagram for the abacus program which computes $\mathrm{Cn}[f, g]$. On the right, we show the contents of the registers just before the adjacent computation is performed. The machine starts with x in register 1 and all other registers empty. p and q are numbers of registers which are not used in the programs for f or g. (Following **BJ**, we often use the word *box* for 'register' in the block diagrams and the text.)



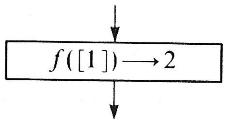
Contents of registers			
$[1]$	$[2]$	$[p]$	$[q]$
x	0	0	0
x	$g(x)$	0	0
x	0	$g(x)$	0
0	0	$g(x)$	x
$g(x)$	0	0	x
$g(x)$	$f(g(x))$	0	x
0	$f(g(x))$	0	x
x	$f(g(x))$	0	0

PROBLEM 1 Given abacus programs for computing the functions f (of 2 arguments) and g_1, g_2 (each of 1 argument) with block diagrams


f([1], [2])⟶3
g₁([1])⟶2
g₂([1])⟶2

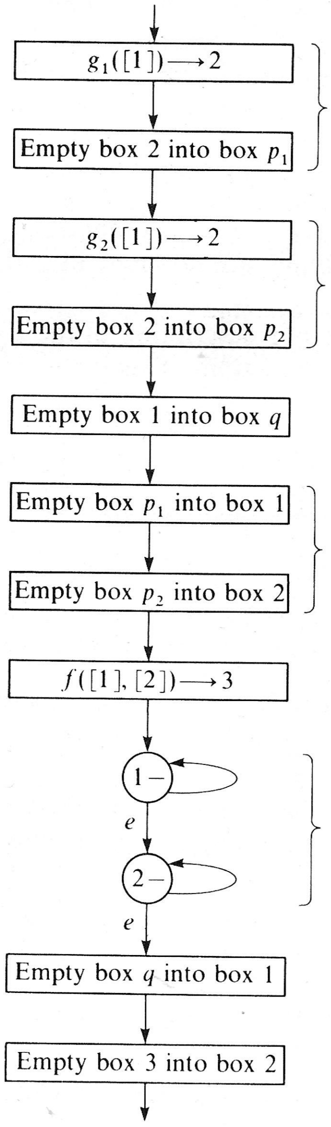
produce an abacus program for computing the function $\mathrm{Cn}[f, g_1, g_2]$.

PROBLEM 2 f is a one argument function which is computed by the abacus with block diagram



The function h satisfies the equations: $h(0) = 0$, $h(s(x)) = f(h(x))$ for all $x \in N$. Prove that h is abacus computable.

SOLUTION 1 We require a program to compute the function $x \longmapsto f(g_1(x), g_2(x))$ for $x \in N$. We simply modify that of **BJ** page 80, Figure 7-2 to take into account the different number of arguments. We must start with x in register 1 (and all other registers empty) and must end up with x in register 1 and $f(g_1(x), g_2(x))$ in register 2. Let p_1, p_2 and q be numbers of registers not used in the programs for f, g_1 or g_2. We shall use registers p_1 and p_2 to store $g_1(x)$ and $g_2(x)$ temporarily, and register q to keep hold of x during the evaluation of $\mathrm{Cn}[f, g_1, g_2](x)$. We must also empty all registers apart from 1 and 2 at the end of the computation. We have drawn the required block diagram below and written explanatory notes alongside.


g₁([1])⟶2
Empty box 2 into box p₁
g₂([1])⟶2
Empty box 2 into box p₂
Empty box 1 into box q
Empty box p₁ into box 1
Empty box p₂ into box 2
f([1], [2])⟶3
1−
e
2−
e
Empty box q into box 1
Empty box 3 into box 2

Work out $g_1(x)$ and put into box p_1

Work out $g_2(x)$ and put into box p_2

Put x into cold storage

Put $g_1(x)$ and $g_2(x)$ into the correct boxes for evaluating f

Evaluate $f(g_1(x), g_2(x))$

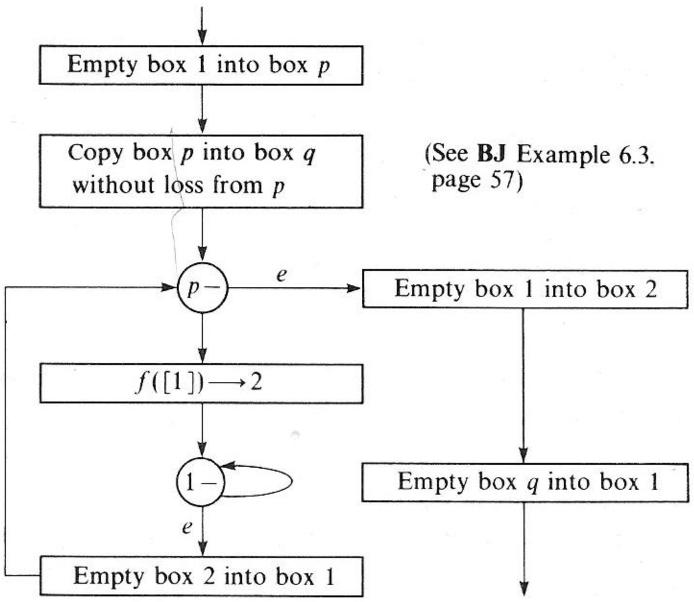
Empty boxes 1 and 2

Bring back x to correct position

Put answer in right box

■

SOLUTION 2 Let p and q be numbers of boxes not used in the program for f. The following program will serve to compute h:



(See **BJ** Example 6.3. page 57)

Suppose we start this computation with x in box 1 and all other boxes empty. First, this program will place x in box p and x in box q and leave all other boxes empty. If $x = 0$, then we shall now exit on the e arrow and halt with all boxes empty, which of course corresponds to $h(0) = 0$. If $x \neq 0$ we shall travel round the loop x times. Each time we reduce $[p]$ by one. The successive contents of box 1 just before we carry out each $p-$ instruction are $0, f(0), f(f(0)), f(f(f(0))), \ldots$, i.e. $h(0), h(1), h(2), h(3), \ldots$. Hence when $[p] = 0$ we shall have $h(x)$ stored in box 1, which is then transferred to box 2, and box q, which contains x, is emptied into box 1. Since all registers apart from 1 and 2 will now be empty, we are done. ■

3.3 Minimization

We need one further operation to complete our description of the class of *recursive* (as opposed to primitive recursive) functions. This is the operation of *minimization*, which is described in the next passage.

READ BJ page 81, line -5 to page 83, line 3.

NOTES (i) *Page 82, line -5*

We can amplify this example as follows. sum is a function of two arguments and so Mn[sum] is the function of one argument defined by:

$$\mathrm{Mn}\,[\mathrm{sum}](x) = \begin{cases} \text{the smallest } y \text{ such that } x + y = 0, \text{ if there is such a } y, \\ \text{undefined otherwise.} \end{cases}$$

Clearly, for $x = 0$, the smallest y such that $x + y = 0$ is $y = 0$, while for $x > 0$ there is no such y—remember we are restricting our attention to natural numbers throughout.

Thus

$$\mathrm{Mn}\,[\mathrm{sum}](x) = \begin{cases} 0, & \text{if } x = 0, \\ \text{undefined otherwise.} \end{cases}$$

(ii) *Page 83, line 2*

Remember z is the zero function here. ▲

EXAMPLE 1 Consider the function f defined by $f(x,y) = x \dot{-} y^2$ for $x, y \in N$. In order to investigate the function $\mathrm{Mn}[f]$, let us note that $x \dot{-} y^2 = 0$ if and only if $y^2 \geq x$. Thus

$$\mathrm{Mn}[f](x) = \text{the smallest natural number } y \text{ such that } y^2 \geq x.$$

So, for example, $\mathrm{Mn}[f](0) = 0$, $\mathrm{Mn}[f](1) = 1$, $\mathrm{Mn}[f](2) = 2$, $\mathrm{Mn}[f](3) = 2$, $\mathrm{Mn}[f](4) = 2$.

EXAMPLE 2 Let f be defined by $f(x,y) = x \dot{-} 2y$ for $x, y \in N$. Then, by definition,

$$\mathrm{Mn}[f](x) = \begin{cases} \text{the smallest } y \text{ for which } x \dot{-} 2y = 0, \text{ if any,} \\ \text{undefined otherwise.} \end{cases}$$

Now $x \dot{-} 2y = 0$ if and only if $x \leq 2y$. Hence $\mathrm{Mn}[f](x) =$ the smallest y for which $2y \geq x$. (Clearly, there always is such a y.)

Thus, if x is even then $\mathrm{Mn}[f](x) = \frac{x}{2}$, and if x is odd then, since $\frac{x}{2}$ is not a natural number, we take the first natural number which is larger, which happens to be $\frac{x+1}{2}$. Thus we have shown

$$\mathrm{Mn}[f](x) = \begin{cases} \dfrac{x}{2}, & \text{if } x \text{ is even,} \\ \dfrac{x+1}{2}, & \text{if } x \text{ is odd.} \end{cases}$$

PROBLEM 1 In each of the following cases, calculate the values of $\mathrm{Mn}[f](0)$, $\mathrm{Mn}[f](2)$, $\mathrm{Mn}[f](5)$.

(i) $f(x,y) = x^2 \dot{-} 3y, \quad x, y \in N$

(ii) $f(x,y) = (x \dot{-} 2y) + (2y \dot{-} x), \quad x, y \in N$

PROBLEM 2 Let $h = \mathrm{Mn}[f]$, where $f(x_1, x_2, y) = x_1 \dot{-} yx_2$. Calculate $h(5,2)$, $h(5,3)$, $h(6,3)$, $h(6,4)$. Describe the function h in familiar terms.

PROBLEM 3 Let $h(x)$ denote the smallest value of y for which $2^y \geq x$. Find a primitive recursive function f (of two arguments) such that $h = \mathrm{Mn}[f]$.

SOLUTION 1 (i) $\mathrm{Mn}[f](0) = 0, \quad \mathrm{Mn}[f](2) = 2, \quad \mathrm{Mn}[f](5) = 9.$

(ii) Although each value of x can be treated separately, we shall describe $\mathrm{Mn}[f]$ explicitly.

Now $f(x,y) = (x \dot{-} 2y) + (2y \dot{-} x) = 0$ if and only if *both* $x \dot{-} 2y = 0$ and $2y \dot{-} x = 0$, i.e. if and only if both $x \leq 2y$ and $2y \leq x$, hence if and only if $x = 2y$. So if x is odd, $f(x,y)$ is never zero, since for *no* natural number y do we have $x = 2y$. If x is even, however, then $f(x,y) = 0$ for exactly one value of y, namely $y = \frac{x}{2}$, so this value must be the smallest possible. Hence

$$\mathrm{Mn}[f](x) = \begin{cases} \dfrac{x}{2} \text{ if } x \text{ is even,} \\ \text{undefined otherwise.} \end{cases}$$

Thus $\mathrm{Mn}[f](0) = 0$, $\mathrm{Mn}[f](2) = 1$ and $\mathrm{Mn}[f](5)$ is undefined. ■

SOLUTION 2 $h(5,2) = 3, \quad h(5,3) = 2, \quad h(6,3) = 2, \quad h(6,4) = 2.$

Now $x_1 \dot{-} yx_2 = 0$ if and only if $x_1 \leq yx_2$, i.e. if and only if $y \geq \frac{x_1}{x_2}$. Hence, if the number x_2 divides x_1, then $\mathrm{Mn}[f](x_1, x_2)$ is the value of the quotient $\frac{x_1}{x_2}$.

Otherwise we can still obtain $\mathrm{Mn}[f](x_1, x_2)$ by dividing x_1 by x_2, forgetting the remainder and adding one to the result. ■

SOLUTION 3 Note that for $x, y \in N$, $2^y \geq x$ if and only if $x \dot{-} 2^y = 0$. Let $f(x, y) = x \dot{-} 2^y$, $x, y \in N$. f is a primitive recursive function (since $\dot{-}$ and $y \longmapsto 2^y \ (y \in N)$ are primitive recursive functions) and $h = \mathrm{Mn}[f]$. ■

We have seen that a function defined by minimization need not be total. If the function obtained from f by minimization does happen to be total, then f is said to be *regular* and the minimization is called *regular minimization.*

READ BJ page 83, line 17 to page 83, line -12.

PROBLEM 4 Which of the examples of a function f given in Examples 1 and 2 and Problems 1 and 2 are regular?

SOLUTION 4 *Example 1* f is regular since for every x there is a y (and therefore a smallest y) such that $y^2 \geq x$, i.e. such that $x \dot{-} y^2 = 0$.

Example 2 This f is regular for similar reasons.

Problem 1 (i) Regular, for similar reasons to those for Example 1 above.

(ii) Not regular, since, for example, there is no y such that $(1 \dot{-} 2y) + (2y \dot{-} 1) = 0$.

Problem 2 Not regular, since if $x_2 = 0$ and $x_1 > 0$, there is no y for which $y \cdot x_2 \geq x_1$. ■

We should now like to define a function to be *recursive* if and only if it can be obtained from the initial functions by a finite number of applications of the operations Cn, Pr and Mn. However, there is one problem with this definition, namely that we have not specified the result of applying these operations to partial functions; and since we know that the operation Mn can lead to partial functions (even when applied to total functions) to which the above definition would allow us to re-apply the operations Cn, Pr and Mn, we must clear up this point.

The case of Cn is easily dealt with. If f is a (possibly partial) function of m arguments, and $g_1, \ldots, g_m$ are (possibly partial) functions of n arguments, we define the function $\mathrm{Cn}[f, g_1, \ldots, g_m]$ as follows:

$$\mathrm{Cn}[f, g_1, \ldots, g_m](x_1, \ldots, x_m) = \begin{cases} f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \\ \quad \text{if } g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n) \text{ are } \textit{all} \\ \quad \text{defined } \textit{and } f \text{ is defined at these values,} \\ \text{undefined otherwise.} \end{cases}$$

Clearly, if f, $g_1, \ldots, g_m$ all happen to be total, then this definition agrees with our old one. Also, a little checking will show that our abacus flow chart for Cn given in Section 3.2 will still work for this extended definition; where a function is not defined the abacus will not halt.

While we could give the extended definitions of Pr and Mn so that the definition of recursive function above would be complete, we do not intend to do so for three reasons. Firstly, they are rather complicated; secondly, we shall never need to apply the operations Pr or Mn to partial functions; and lastly, we shall prove in *Unit 4* that these extended operations are unnecessary anyway; that is, they produce no functions that cannot already be obtained with the operations we have met so far.

Thus, for our purposes, it is sufficient to define a function to be *recursive* if and only if it can be obtained from the initial functions by a finite number of applications of the operations Cn, Pr and Mn, where Pr and Mn are applied only to total functions.

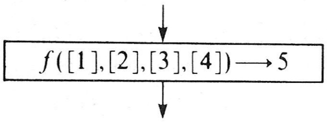
We conclude this unit with a proof that all recursive functions are abacus computable. We have shown already that the initial functions are abacus computable, and that the abacus computable functions are closed under the operations Cn and Pr. Hence it remains to show only that the abacus computable functions are closed under the operation Mn. That is, we must show that given any (total) abacus computable function f, $\mathrm{Mn}[f]$ is also abacus computable.

It is not difficult to see how an abacus can perform minimization. If $h = \mathrm{Mn}[f]$, where f is a function of $n + 1$ arguments, then to calculate $h(x_1, \ldots, x_n)$, the machine needs to compute successively the values $f(x_1, \ldots, x_n, 0), f(x_1, \ldots, x_n, 1), f(x_1, \ldots, x_n, 2), \ldots$. If none of these values is 0, the machine never completes its computation. Otherwise, the machine stops when it finds the first y which makes $f(x_1, \ldots, x_n, y)$ equal to 0 and produces this y as the result of the computation. Such an abacus clearly computes the function h. Notice that if the abacus does not halt for some input, then h will not be defined for the corresponding arguments, and vice versa. This corresponds exactly to our definition of an abacus computable *partial* function.

READ BJ page 83, line −11 to page 84, line 7.

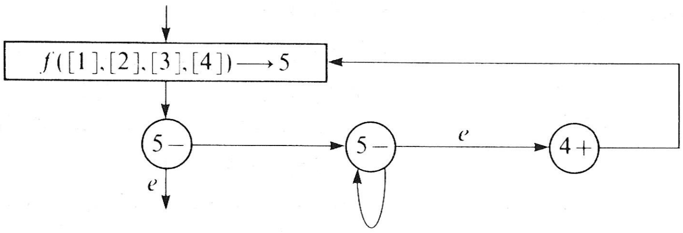
NOTE *Page 83, line −5*
In fact, initially *all* the boxes apart from box 1 are empty, and box 1 contains x—remember we are computing here the function h, which has only one argument, so we want to end up with x in box 1 and (if $h(x)$ is defined) $h(x)$ in box 2. ▲

PROBLEM 5 Given a program



for computing a function f of 4 arguments, draw the flow graph for the machine which computes $\mathrm{Mn}[f]$. ($\mathrm{Mn}[f]$ is a three argument function, so construct your flow graph so that the arguments are initially in registers 1, 2, 3 and the answer (if defined) appears in register 4.)

SOLUTION 5



■

3.4 Objectives of Unit 3

We list those things on which we may set assessment questions to test your understanding of the unit. After working through the unit, we feel you should be able to:

(i) explain what is meant by a primitive recursive function;

(ii) compute the values of a primitive recursive function given its definition;

(iii) understand the proofs that the functions sum, prod, pr, tpl, pred, dif, adf, sg, $\overline{\mathrm{sg}}$, and those given in Problems 11, 12, 14, 15 and 16 of Section 3.1 are all primitive recursive;

(iv) prove that a simple function is primitive recursive by defining it using the operations of composition and primitive recursion in terms of *either* the basic functions, i.e. the zero, successor and identity functions, *or* other given primitive recursive functions;

(v) explain what is meant by a recursive function, i.e. one whose definition includes use of minimization;

(vi) compute the values of a recursive function whose definition includes use of minimization;

(vii) explain in detail how the zero, successor and identity functions and operations of composition and primitive recursion can be carried out on an abacus;

(viii) explain in outline the proof that all recursive functions are abacus computable.

UNIT 4
CHURCH'S THESIS

4.0 Introduction

In *Units 1–3*, we described three classes of functions: the class of Turing computable functions, which we denote by T, the class of abacus computable functions, which we denote by A, and the class of recursive functions, which we denote by R. The functions in these classes are certainly computable in the intuitive sense of the word. However, if it were the case that two of these classes were different, then we would have the problem of deciding which of the classes, if any, provided the most natural precise definition of the intuitive notion of 'computable'. Fortunately, the three classes turn out to be the same, which it is the aim of this unit to prove.

Since we have shown already that $A \subseteq T$ (*Unit 2*) and $R \subseteq A$ (*Unit 3*), we need only show $T \subseteq R$, for we shall then have the inclusions $T \subseteq R \subseteq A \subseteq T$, which of course imply that the three classes are the same. Thus we must prove that every Turing computable function is recursive. Not surprisingly, the proof will require us to have a good stock of functions on hand which are known to be recursive, so in the next section we provide many more examples of recursive functions. We then present the proof that $T \subseteq R$ and conclude the unit with a discussion on Church's thesis in the light of the results we have obtained so far in the course.

4.1 Further Properties and Examples of Recursive Functions

The aim of this section is to show that a wide class of functions is recursive. We do this not only by constructing recursive definitions of given functions, but also by showing that the class of recursive functions is closed under various natural operations, just as we showed it was closed under addition and multiplication of functions, in Problem 12 of Section 3.1 of *Unit 3*. (Strictly speaking, that problem was posed in terms of primitive recursive functions, but the proof is just the same for recursive functions.)

The next reading passage begins with a summary of the definitions of some primitive recursive functions that we have met already and then discusses an operation on functions known as 'definition by cases', which requires some introductory remarks.

By a *condition* on pairs (x, y) of natural numbers, we mean some well-defined relation between x and y. For example, '$x = y$', '$x \leq y$', '$x + y^2 = 2xy$' are all conditions on pairs (x, y). We say a given pair (x, y) *satisfies* a given condition C if the relation corresponding to C holds between x and y. Thus, $(3, 3)$ does satisfy '$x \geq y$', but $(3, 5)$ does not. A collection $C_1, \ldots, C_n$ of conditions is called *mutually exclusive* if every pair (x, y) of natural numbers satisfies at most one of the conditions, and the collection is called *collectively exhaustive* if every pair satisfies at least one of the conditions. Thus the collection '$x < y$', '$y < x$' is mutually exclusive (as for no natural numbers x and y can we have both $x < y$ and $y < x$ true), while the collection '$x < y$', '$x = y$', '$x > y$' is not only mutually exclusive but also collectively exhaustive (because for all pairs of natural numbers x, y, exactly one of $x < y$, $x = y$, $x > y$ is true).

While the reading passage below concentrates on conditions on pairs, we remark that all the above definitions have obvious generalizations to triples, quadruples, and so on, of natural numbers, and to conditions on natural numbers singly; e.g. 'x is even' is a condition on natural numbers x.

ROBLEM 1 Which of the following pairs of natural numbers satisfy the condition '$x \geq 3y$'?
(i) $(4, 1)$, (ii) $(0, 0)$, (iii) $(8, 3)$.

PROBLEM 2 For each of the following collections of conditions, state whether it is (i) mutually exclusive, and (ii) collectively exhaustive.

(a) '$x \geq 2y$', '$x = 2y$', '$x \leq 2y$'.

(b) '$x > 2y$', '$x < 2y$'.

(c) '$x + y = 10$', '$x \geq y$'.

SOLUTION 1 (i) and (ii) do satisfy the condition; (iii) does not. ■

SOLUTION 2 (a) Not mutually exclusive (e.g. $(2, 1)$ satisfies all three conditions), but collectively exhaustive.

(b) Mutually exclusive, but not collectively exhaustive (e.g. $(2, 1)$ satisfies neither condition).

(c) Neither mutually exclusive (e.g. $(7, 3)$ satisfies both conditions), nor collectively exhaustive (e.g. $(0, 11)$ satisfies neither condition). ■

READ BJ page 84, line 8 to page 85, line −1.

NOTES (i) *Page 85, line 10*

Clearly, c_i is uniquely determined by the condition C_i. Thus we usually speak of *the* characteristic function of a condition. We have met already some examples of characteristic functions, in *Unit 2*. In Example 4 of Section 2.2 of *Unit 2*, we showed that the characteristic function of the condition '$x > y$' (i.e. the function f of this example) is abacus computable. Similarly, the functions e, d, and prime of Problem 16(iii), (iv), (v) of Section 2.2 of *Unit 2* are the characteristic functions of the conditions '$x = y$', 'x is divisible by y' and 'x is prime', respectively.

(ii) *Page 85, line 16 (box)*

This function can be shown to be primitive recursive, by repeated application of the results of Problem 12 of Section 3.1 of *Unit 3*. ▲

From now on we shall call a condition *primitive recursive* if its characteristic function is a primitive recursive function.

EXAMPLE 1 We show that the condition '$x = y$' is primitive recursive.

First, notice that $\mathrm{adf}(x, y) = |x - y| = 0$ if $x = y$, and $\mathrm{adf}(x, y) > 0$ if $x \neq y$. So if $x = y$, then $\overline{\mathrm{sg}}\,(\mathrm{adf}(x, y)) = \overline{\mathrm{sg}}\,(0) = 1$ and if $x \neq y$ then $\overline{\mathrm{sg}}\,(\mathrm{adf}(x, y)) = 0$. Thus the function Eq, defined by $\mathrm{Eq}(x, y) = \overline{\mathrm{sg}}\,(\mathrm{adf}(x, y))$ is the characteristic function of the condition '$x = y$', and Eq is clearly primitive recursive, so the condition '$x = y$' is primitive recursive.

EXAMPLE 2 If C is any condition, then the condition 'not C' is defined in the obvious way, namely, a pair (x, y) satisfies 'not C' if and only if (x, y) does *not* satisfy C. (For example, if C is the condition '$x \geq 2y$', then 'not C' is the condition '$x < 2y$'.) If c is the characteristic function of C, then the function c^* defined by $c^*(x, y) = \overline{\mathrm{sg}}\,(c(x, y))$ will be the characteristic function of 'not C'. We see this as follows. First, if (x, y) does satisfy 'not C' then (x, y) does *not* satisfy C, so $c(x, y) = 0$: then $c^*(x, y) = \overline{\mathrm{sg}}\,(c(x, y)) = \overline{\mathrm{sg}}\,(0) = 1$. Second, if (x, y) does not satisfy 'not C', then (x, y) *does* satisfy C, so $c(x, y) = 1$; but then $c^*(x, y) = \overline{\mathrm{sg}}\,(1) = 0$.

So if C is a primitive recursive condition, c is a primitive recursive function, and $c^* = \mathrm{Cn}\,[\overline{\mathrm{sg}}, c]$ is also a primitive recursive function. Hence 'not C' is a primitive recursive condition.

PROBLEM 3 Prove that the condition '$x > 2y$' on pairs (x, y) is primitive recursive.

PROBLEM 4 If C_1, C_2 are any conditions, the condition 'C_1 and C_2' is defined as follows: any (x, y) satisfies 'C_1 and C_2' if and only if (x, y) satisfies C_1 and (x, y) satisfies C_2. Prove that if C_1, C_2 are both primitive recursive then so is 'C_1 and C_2'.

PROBLEM 5 Suppose that $C_1, \ldots, C_n$ is a mutually exclusive (but not necessarily collectively exhaustive) collection of primitive recursive conditions on pairs (x, y), and that $g_1, \ldots, g_n, g_{n+1}$ are primitive recursive functions of two arguments. The function f is defined by

$$f(x,y) = \begin{cases} g_1(x,y), & \text{if } C_1, \\ \vdots & \quad\vdots \\ g_n(x,y), & \text{if } C_n, \\ g_{n+1}(x,y), & \text{otherwise (i.e. if } (x,y) \text{ satisfies} \\ & \text{none of the conditions } C_1, \ldots, C_n). \end{cases}$$

Prove that f is primitive recursive.

PROBLEM 6 Let n_0, m_0 be any fixed natural numbers. Prove that the condition '$x = n_0$ and $y = m_0$' is primitive recursive.

PROBLEM 7 In the next section we shall need the function a defined by:

$$a(x,y) = \begin{cases} 1, & \text{if } x = 1 \text{ and } y = 0, \\ 3, & \text{if } x = 1 \text{ and } y = 1, \\ 2, & \text{if } x = 2 \text{ and } y = 1, \\ 2, & \text{if } x = 3 \text{ and } y = 1, \\ y, & \text{otherwise.} \end{cases}$$

Prove that a is primitive recursive.

SOLUTION 3 The characteristic function of '$x > 2y$' is the function c defined by $c(x, y) = \text{sg}(x \dot{-} 2y)$. c is primitive recursive, since it is a composition of known primitive recursive functions. ■

SOLUTION 4 Let c_1, c_2 be the characteristic functions of C_1, C_2 respectively. We are given that c_1, c_2 are primitive recursive. We require a function c such that for all numbers x and y, $c(x, y) = 1$ if $c_1(x, y) = c_2(x, y) = 1$ and $c(x, y) = 0$ if $c_1(x, y) = 0$ or $c_2(x, y) = 0$ (or both). Further, c should be primitive recursive.

Define $c = c_1 \cdot c_2$ (i.e. $c(x, y) = c_1(x, y) \cdot c_2(x, y)$ for all x, y). c is primitive recursive by Problem 12 of Section 3.1 of *Unit 3*, and clearly has the required properties. ■

SOLUTION 5 That f is primitive recursive will follow from 'definition by cases' if we can show that the condition 'otherwise' is primitive recursive. Now the pair (x, y) satisfies this condition if and only if (x, y) satisfies none of the conditions $C_1, \ldots, C_n$, that is, if and only if (x, y) satisfies all the conditions 'not C_1', …, 'not C_n'. Thus we may write the condition 'otherwise' as 'not C_1 and not C_2 and … and not C_n'. Example 2 and repeated application of Problem 4 above show that this condition is primitive recursive. ■

SOLUTION 6 Define the function c by $c(x, y) = \text{Eq}(n_0, x) \cdot \text{Eq}(m_0, y)$, where Eq is the function of Example 1 above. Then c is clearly the characteristic function of the condition '$x = n_0$ and $y = m_0$', and is primitive recursive since it is a composition of known primitive recursive functions. (We have also used here Problem 9(i) of Section 3.1 of *Unit 3*.) ■

SOLUTION 7 Define the functions $g_1, \ldots, g_5$ by: $g_1(x, y) = 1$, $g_2(x, y) = 3$, $g_3(x, y) = 2$, $g_4(x, y) = 2$, $g_5(x, y) = y$ $(= \text{id}_2^2(x, y))$. Then $g_1, \ldots, g_5$ are easily seen to be primitive recursive. Further, the first four conditions appearing in the definition of a are primitive recursive by Problem 6 above, and clearly form a mutually exclusive collection. Hence a is primitive recursive by Problem 5 above. ■

The results contained in the following examples and problems will be required in the next two sections of this unit. We therefore ask you to study them carefully even if you do not have enough time to attempt all the problems.

EXAMPLE 3 Suppose f is a function of $n + 1$ arguments. We say the function h is defined from f by *general summation* if, for all natural numbers $x_1, \ldots, x_n, y$ we have

$$\begin{aligned} h(x_1, \ldots, x_n, y) &= \sum_{z=0}^{y} f(x_1, \ldots, x_n, z) \\ &= f(x_1, \ldots, x_n, 0) + f(x_1, \ldots, x_n, 1) + \cdots + f(x_1, \ldots, x_n, y). \end{aligned}$$

We show that h is primitive recursive whenever f is.

The argument is very similar to that in Problem 11 of Section 3.1 of *Unit 3*, except that there f was a function of just one argument and we summed the numbers $f(z)$ only for $z < y$. Here we sum the values of $f(x_1, \ldots, x_n, z)$ for $z \le y$, so in particular we have $h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n, 0)$. Define g by $g(x_1, \ldots, x_n) = f(x_1, \ldots, x_n, 0)$. Then g is primitive recursive by an obvious extension of Problem 9(i) of Section 3.1 of *Unit 3*.

We shall find a primitive recursive function j such that

$$\begin{aligned} &h(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n), \\ &h(x_1, \ldots, x_n, s(y)) = j(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)). \end{aligned}$$

Then as these equations are in the correct format of primitive recursion, we shall have $h = \mathrm{Pr}[g, j]$, so h is primitive recursive.

So we define j by $j(x_1, \ldots, x_n, y, u) = u + f(x_1, \ldots, x_n, s(y))$. Then j is a composition of primitive recursive functions and therefore primitive recursive.

EXAMPLE 4 We show next that the condition 'x is divisible by y' is primitive recursive. We denote the characteristic function of this condition by d. You have already met d in Problem 16(iv) of Section 2.2 of *Unit 2*, defined by

$$d(x, y) = \begin{cases} 1, & \text{if } x \text{ is divisible by } y, \\ 0, & \text{if } x \text{ is not divisible by } y. \end{cases}$$

To show d primitive recursive is the trickiest problem we have met so far and we use the result of Example 3 above.

First let x, y be any natural numbers and consider the sequence of numbers

$$\mathrm{Eq}(x, 0 \cdot y), \mathrm{Eq}(x, 1 \cdot y), \mathrm{Eq}(x, 2 \cdot y), \ldots, \mathrm{Eq}(x, x \cdot y),$$

where Eq is the function of Example 1 above.

Now if x *is* divisible by y, then there will be a unique number z such that $x = z \cdot y$. Obviously, such a z will satisfy $z \le x$ ($z = x$ is possible if $y = 1$), so exactly one member of the above sequence will equal 1 (namely $\mathrm{Eq}(x, z \cdot y)$) and the rest will equal 0. On the other hand, if x is *not* divisible by y, then all numbers of the sequence will equal 0. Hence, if we add up all these numbers, their sum will be 1 if x is divisible by y and 0 if x is not divisible by y.

So we define d by $d(x, y) = \sum_{z=0}^{x} \mathrm{Eq}(x, z \cdot y)$.

To show d is primitive recursive is now an easy matter. First define f by $f(x, y, z) = \mathrm{Eq}(x, z \cdot y)$: f is clearly primitive recursive. Hence, by Example 3 above, so is the function h defined by $h(x, y, t) = \sum_{z=0}^{t} f(x, y, z)$. But for all x and y, $d(x, y) = h(x, y, x)$. Thus $d = \mathrm{Cn}[h, \mathrm{id}_1^2, \mathrm{id}_2^2, \mathrm{id}_1^2]$, so d is primitive recursive.

EXAMPLE 5 Our last example in this section is the function lo defined by

$$\mathrm{lo}(x) = \begin{cases} \text{the largest number } w \text{ such that } 2^w \leq x, & \text{if } x > 0, \\ 0, & \text{if } x = 0. \end{cases}$$

The second clause is needed here since there is no natural number w such that $2^w \leq 0$. The first few values of lo are given in the following table.

$x =$	0	1	2	3	4	5	6	7	8	9...
$\mathrm{lo}(x) =$	0	0	1	1	2	2	2	2	3	3...

(Notice that $\mathrm{lo}(x)$ equals the integer part of $\log_2 x$, hence the name.)

We use a similar technique to Example 4 above in order to show that lo is primitive recursive. Here we let x be any positive natural number and consider the sequence of numbers

$$\mathrm{sg}(s(x) \dot{-} 2^1), \mathrm{sg}(s(x) \dot{-} 2^2), \ldots, \mathrm{sg}(s(x) \dot{-} 2^x).$$

Let $w = \mathrm{lo}(x)$. Then $2^1, 2^2, \ldots, 2^w$ are all less than or equal to x, and $2^{w+1}, 2^{w+2}, \ldots, 2^x$ are all greater than x, by definition of the function lo. (It is easy to show that $\mathrm{lo}(x) < x$ for all positive x.) So $s(x) \dot{-} 2^t > 0$ if $t \leq w$, and $s(x) \dot{-} 2^t = 0$ if $t > w$. Hence the first w numbers in the above sequence are 1 and the rest are 0. Hence

$$\mathrm{lo}(x) = w = \sum_{z=1}^{x} \mathrm{sg}(s(x) \dot{-} 2^z).$$

In order to apply Example 3 above, we must add in the term for $z = 0$ which is $\mathrm{sg}(s(x) \dot{-} 2^0) = \mathrm{sg}(s(x) \dot{-} 1) = 1$ (since for the moment we are considering only $x \geq 1$, so $s(x) \geq 2$).

Hence $\mathrm{lo}(x) = \left(\sum_{z=0}^{x} \mathrm{sg}(s(x) \dot{-} 2^z)\right) \dot{-} 1$. However, we have derived this formula assuming only $x > 0$; but notice that the right-hand side is 0 when $x = 0$ so the formula holds good for all values of x. It follows immediately that lo is primitive recursive, using Example 3 above and Problem 16 of Section 3.1 of *Unit 3* (which tells us that the function $z \longmapsto 2^z$ $(z \in N)$ is primitive recursive).

PROBLEM 8 Suppose f is a primitive recursive function of $n + 1$ arguments and that the function h is defined by

$$h(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) \cdot f(x_1, \ldots, x_n, 1) \cdot \ldots \cdot f(x_1, \ldots, x_n, y).$$

(This function is called the *general product*, and the right-hand side of the above equation is often written as $\prod_{z=0}^{y} f(x_1, \ldots, x_n, z)$.)

Prove that h is primitive recursive.

PROBLEM 9 Let j be defined by

$$j(x) = \begin{cases} \dfrac{x}{2}, & \text{if } x \text{ is divisible by 2}, \\ 0, & \text{if } x \text{ is not divisible by 2}. \end{cases}$$

Prove that j is primitive recursive.

ROBLEM 10 The functions lft, ctr and rgt are defined as follows.

$$\mathrm{lft}(x) = \begin{cases} \text{the largest number } w \text{ such that } 2^w \text{ divides } x, & \text{if } x > 0, \\ 0, & \text{if } x = 0; \end{cases}$$

$$\mathrm{ctr}(x) = \begin{cases} \text{the largest number } w \text{ such that } 3^w \text{ divides } x, & \text{if } x > 0, \\ 0, & \text{if } x = 0; \end{cases}$$

$$\text{rgt}(x) = \begin{cases} \text{the largest number } w \text{ such that } 5^w \text{ divides } x, & \text{if } x > 0, \\ 0, \quad \text{if } x = 0. \end{cases}$$

Prove that lft, ctr, rgt are primitive recursive. (The names of the functions are meant to suggest 'left', 'centre' and 'right', which will be of significance later in this unit.)

SOLUTION 8 The proof is analogous to Example 3 above. We define g by $g(x_1, \ldots, x_n) = f(x_1, \ldots, x_n, 0)$: g is primitive recursive, by a suitable adaptation of Problem 9(i) of Section 3.1 of *Unit 3*; and we define j by

$$j(x_1, \ldots, x_n, y, u) = u \cdot f(x_1, \ldots, x_n, s(y)).$$

j is clearly primitive recursive and h satisfies the equations

$$h(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n),$$

$$h(x_1, \ldots, x_n, s(y)) = j(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)).$$

Thus $h = \text{Pr}[g,j]$ and is therefore primitive recursive. ■

SOLUTION 9 In Problem 15 of Section 3.1 of *Unit 3*, we showed that the function h is primitive recursive, where h is defined by

$$h(x) = \begin{cases} \dfrac{x}{2}, & \text{if } x \text{ is even}, \\ \dfrac{x-1}{2}, & \text{if } x \text{ is odd}. \end{cases}$$

Effectively, we want the first part of this function but not the second, so we shall aim to multiply h by a function whose value is 1 when x is even and 0 otherwise. Now in Example 4 above we showed that the function d, where

$$d(x,y) = \begin{cases} 1, & \text{when } x \text{ is divisible by } y, \\ 0, & \text{otherwise}, \end{cases}$$

is primitive recursive. This can be adapted by putting $y = 2$ and using a result similar to that of Problem 9(i) of Section 3.1 of *Unit 3* to give a primitive recursive function d' where

$$d'(x) = \begin{cases} 1, & \text{when } x \text{ is divisible by 2, i.e. is even}, \\ 0, & \text{otherwise}. \end{cases}$$

So we can define j by $j(x) = h(x) \cdot d'(x)$, which as a product of primitive recursive functions is itself primitive recursive (by Problem 12 of Section 3.1 of *Unit 3*). ■

SOLUTION 10 The technique required is similar to that used in Examples 4 and 5 above. Let x be any positive number and consider

$$d(x, 2^1), d(x, 2^2), \ldots, d(x, 2^x),$$

where d is the characteristic function of the condition 'x is divisible by y'.

Let $w = \text{lft}(x)$. Then x is divisible by 2^t for $t \leq w$, and is not divisible by 2^t for $t > w$. Thus the first w numbers in the sequence above are 1, and the rest are 0. (We clearly have $w < x$.) Hence

$$\text{lft}(x) = w = \sum_{z=1}^{x} d(x, 2^z)$$

$$= \left(\sum_{z=0}^{x} d(x, 2^z) \right) \dot{-} 1 \quad (\text{since } d(x, 2^0) = d(x, 1) = 1).$$

As before, we derived this formula on the assumption that $x > 0$, but it holds for $x = 0$ as well, as you may easily check.

By similar arguments we have

$$\mathrm{ctr}(x) = \left(\sum_{z=0}^{x} d(x, 3^z)\right) \dot{-} 1$$

and

$$\mathrm{rgt}(x) = \left(\sum_{z=0}^{x} d(x, 5^z)\right) \dot{-} 1.$$

As in Example 4, we can now conclude that lft, ctr, rgt are primitive recursive. (Note that $3^z = \exp(3, z)$ and $5^z = \exp(5, z)$.) ■

4.2 Recursive Functions Which Describe Turing Machines

In this section we begin the proof that all Turing computable functions are recursive. The basic idea is to use a single number to describe or 'code' the configuration of a Turing machine at any given moment in the course of its computation. As the computation proceeds and the configuration changes, the number which describes the configuration also changes. So for a given Turing machine, M, with given input, x, there is a well-defined function whose value for argument t is the number which codes the configuration of M at stage t of its computation with input x. The crux of the proof is to show that this function is recursive.

The configuration of a Turing machine at any stage of the computation is given by:

(i) the symbols on the tape;

(ii) the tape square currently being scanned;

(iii) the machine's current state.

The information under (i) and (ii) can equally well be given by specifying

(i)′ the symbols on the tape to the left of the scanned square;

(ii)′ the symbols on the tape on the scanned square and to its right.

The next reading passage shows how these two pieces of information can be described by two numbers, which we will call the *left number* and the *right number*. This description makes use of binary notation for numbers, which we now briefly describe.

Binary Notation

Our usual decimal notation relies on the fact that every natural number can be expessed in only one way as a combination of powers of 10, with coefficients chosen from the range $0 \leq n \leq 9$. For example,

$$5204 = \underline{5} \cdot 10^3 + \underline{2} \cdot 10^2 + \underline{0} \cdot 10^1 + \underline{4} \cdot 10^0.$$

The coefficients in this representation, 5, 2, 0, 4, make up the decimal notation for the given number.

In exactly the same way, every natural number can also be expressed in only one way as a combination of powers of 2, with coefficients in the range $0 \leq n \leq 1$. For example

$$179 = \underline{1} \cdot 2^7 + \underline{0} \cdot 2^6 + \underline{1} \cdot 2^5 + \underline{1} \cdot 2^4 + \underline{0} \cdot 2^3 + \underline{0} \cdot 2^2 + \underline{1} \cdot 2^1 + \underline{1} \cdot 2^0.$$

The coefficients in this expression are 1, 0, 1, 1, 0, 0, 1, 1 and these make up the *binary numeral* for the number 179. Thus the binary numeral for 179 is 10110011. As other examples, note that

$$7 = 1\cdot 2^2 + 1\cdot 2 + 1\cdot 2^0$$

$$20 = 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2 + 0\cdot 2^0$$

so the binary numeral for 7 is 111, and for 20 is 10100.

The binary numeral for a number n is easily calculated. We successively divide n by 2 until we reach 0, and write down the successive remainders from *right* to *left*. Thus for $n = 75$ we have

2)75
2)37 remainder 1
2)18 remainder 1
2) 9 remainder 0
2) 4 remainder 1
2) 2 remainder 0
2) 1 remainder 0
1 remainder 1

so the binary numeral for 75 is 1001011.

We shall need the following easily verifiable properties of binary numerals:

Property 1 2^m has binary numeral $1\underbrace{00\ldots0}_{m\,0\text{s}}$, for any number m.

Property 2 The last (i.e. right-most) digit of the binary numeral for a number n (referred to by **BJ** as the 'one's place') is 0 if and only if n is even.

Property 3 $2^m - 1$ has binary numeral $\underbrace{11\ldots1}_{m\,1\text{s}}$ for any number m. (Set $x = 2$ in the identity $x^m - 1 = (x-1)(x^{m-1} + x^{m-2} + \cdots + x + 1)$.)

PROBLEM 1 Which numbers (giving your answers in decimal notation!) have binary numerals

(i) 1001, (ii) 111101?

PROBLEM 2 Find the binary numerals denoting

(i) 23, (ii) 62.

PROBLEM 3 Describe the binary numeral for the number $(2^m - 1)2^n$, where m and n are any given numbers.

PROBLEM 4 Which number has binary numeral $\underbrace{11\ldots1}_{m\,1\text{s}}0\underbrace{11\ldots1}_{p\,1\text{s}}$, where m and p are any given numbers?

PROBLEM 5 If the last digit of a binary numeral is changed, what is the effect on the corresponding decimal number?

SOLUTION 1 (i) $1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2 + 1 = 9$

(ii) $1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2 + 1 = 61$ ■

SOLUTION 2 (i) 10111, (ii) 111110. ■

SOLUTION 3 $\underbrace{11\ldots1}_{m\,1\text{s}}$ represents $2^m - 1$, and adding n 0s on the right of these 1s has the effect of multiplying by 2^n. Hence the required binary numeral is $\underbrace{11\ldots1}_{m\,1\text{s}}\underbrace{00\ldots0}_{n\,0\text{s}}$. ■

SOLUTION 4 The required number is

$$2^{m+p} + 2^{m+p-1} + \cdots + 2^{p+1} + 2^{p-1} + 2^{p-2} + \cdots + 2^1 + 2^0$$

$$= (2^m - 1)2^{p+1} + (2^p - 1) \quad \text{(by Problem 3 and Property 3).} \ ■$$

SOLUTION 5 Clearly, changing a last 0 to 1 results in adding 1 to the corresponding number and changing a last 1 to 0 results in subtracting 1 from the corresponding number. ■

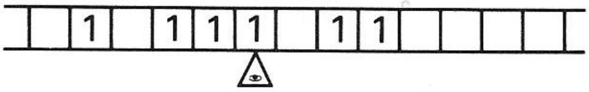
READ BJ page 89, line 1 to page 90, line −10.

NOTE *Page 89, line 17*
Remember that we are now using the convention that the number n is represented by an unbroken block of $(n + 1)$ 1s. ▲

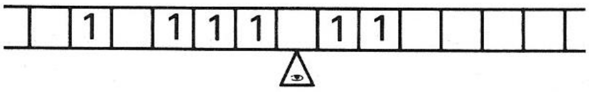
NOTATION We shall use l and r to denote respectively the left and right numbers of the configuration of the tape.

PROBLEM 6 Calculate l and r in each of the following cases.

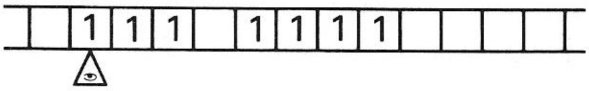
(i)



(ii)



(iii)



(All tape squares not shown are blank.)

PROBLEM 7 Draw the configuration of the tape in each of the following cases;

(i) $l = 5$, $r = 15$; (ii) $l = 11$, $r = 8$; (iii) $l = 20$, $r = 0$.

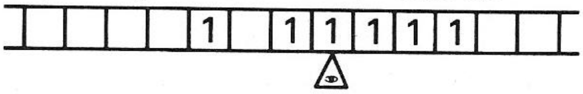
SOLUTION 6 (i) The left numeral = 1011; the right numeral = 1101. So $l = 2^3 + 2 + 1 = 11$, $r = 2^3 + 2^2 + 1 = 13$.

(ii) The left numeral = 10111; the right numeral = 110. So $l = 2^4 + 2^2 + 2 + 1 = 23$, $r = 2^2 + 2 = 6$.

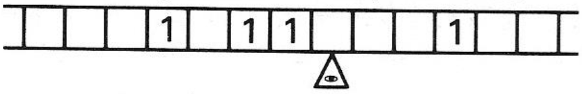
(iii) The left numeral = 0; the right numeral = 11110111. So $l = 0$, $r = 2^7 + 2^6 + 2^5 + 2^4 + 2^2 + 2 + 1 = 247$. ■

SOLUTION 7

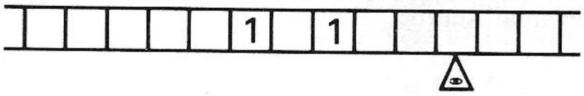
(i) The left numeral = 101; the right numeral = 1111; hence the configuration is



(ii) The left numeral = 1011; the right numeral = 1000; hence the configuration is



(iii) The left numeral = 10100; the right numeral = 0; hence the configuration is



■

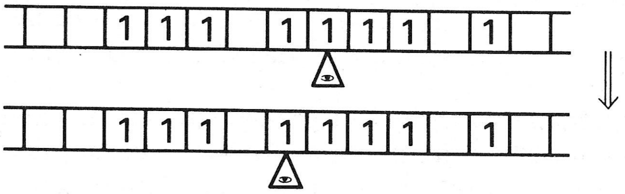
We now investigate how l and r change as the result of a single step of M's computation. There are, of course, four steps to consider: the scanned symbol changing from 0 to 1 or from 1 to 0, and M moving one square to the left or to the right.

READ BJ page 90, line −9 to page 91, line 16.

NOTES (i) *Page 90, line −8*
The effect on the right number stated here follows from Problem 5 above.

(ii) *Page 91, line 1*
See Property 2 of binary notation above.

(iii) *Page 91, lines 2–12*
We illustrate the change in the numerals in the continuation of **BJ** Example 8.1. When the machine moves left, the effect on the right numeral is to place a 1 at its right-hand end. (Remember that after the move, the machine will be scanning a 1, and that the right numeral is the sequence on the tape to the right of, and including, the scanned symbol *written backwards.*) So for example, 10111 changes to 101111, corresponding to the move below.



To see why this changes r to $2r + 1$ we split the change in the right numeral into two stages: (i) place a 0 at the right-hand end; (ii) change this 0 to a 1. The stages in our example above are

$$10111 \longrightarrow 101110 \longrightarrow 101111.$$

Now it is easy to see that the effect on r resulting from stage (i) is to multiply it by 2, and the effect of stage (ii) is to add 1 (see Problem 5 above). Thus the total change in r is $r \longmapsto 2r \longmapsto 2r + 1$. Similarly, the effect on the left numeral may be split into first changing its last 1 to a 0, which changes l to $l - 1$ (by Problem 5), and then removing this zero, which clearly changes $l - 1$ to $\frac{l-1}{2}$. ▲

PROBLEM 8 Prove that if M moves right and r is even, then $l' = 2l$ and $r' = \frac{r}{2}$ (**BJ** page 91, line 16).

PROBLEM 9 In each of the following cases, calculate the new values of l and r when the machine moves (a) left and (b) right:

(i) $l = 15, r = 25$; (ii) $l = 8, r = 25$.

SOLUTION 8 Since r is even, the scanned symbol before the move is 0 (by Property 2 of binary notation). Hence the effect of the move is to place a 0 at the end of the left numeral and to remove a 0 from the end of the right numeral. Thus the original l is multiplied by 2, and the original r is divided by 2, so $l' = 2l$ and $r' = \frac{r}{2}$. ■

SOLUTION 9 (i) (a) For a left move, since l is odd we apply **BJ**'s first case so
$$l' = \frac{l-1}{2} = \frac{15-1}{2} = 7 \text{ and } r' = 2r + 1 = 2 \cdot 25 + 1 = 51.$$

(b) For a right move, since r is odd we apply **BJ**'s third case, so
$$l' = 2l + 1 = 31 \text{ and } r' = \frac{r-1}{2} = 12.$$

(ii) (a) For a left move, since l is even we apply **BJ**'s second case, so $l' = \frac{l}{2} = 4$ and $r' = 2r = 50$.

(b) For a right move, since r is odd we apply **BJ**'s third case, so
$$l' = 2l + 1 = 17 \text{ and } r' = \frac{r-1}{2} = 12. \quad ■$$

The next reading passage shows that there is a primitive recursive function of two arguments x_1, x_2, which gives the value of r when the machine is in the initial configuration for the calculation of $f(x_1, x_2)$, and that there is also a primitive recursive function which gives the value of $f(x_1, x_2)$ in terms of the value of r when the machine halts.

READ BJ page 91, line 17 to page 92, line 3.

NOTES (i) *Page 91, lines -13 to -11*
We met these results in Property 3 of binary notation and Problems 3 and 4 above.

(ii) *Page 91, line -9*
Notice that **BJ** is now using $\dot{-}$ signs instead of minus signs. This is allowable since $2^m \geq 1$ for $m \in N$, and it lets us deduce that $s(x_1, x_2)$ is a primitive recursive function, as it is now the composition of functions shown to be primitive recursive in *Unit 3*. (Be careful not to confuse this function s of two arguments with the successor function, which, of course, has only one argument.)

(iii) *Page 91, lines -5 to -3*
This follows from Property 3 of binary notation.

(iv) *Page 91, line -1*
Ignore the comment in parentheses here. Our proof that lo is primitive recursive appears in Section 4.1 as Example 5. ▲

Remember that the aim of this section is to show that if f is a Turing computable function (of two arguments) then it is recursive. By now we know we can express the value of $f(x_1, x_2)$ as $\mathrm{lo}(r)$, where r is the right number when the machine stops. Our problem now is to find a recursive way of describing r and some way of knowing when the computation is finished. As a first step in this direction, we shall find out how to specify any Turing machine recursively.

We already know that the behaviour of a Turing machine is completely determined by its machine table or its flow graph. **BJ** now shows how to describe the flow graph by a pair of primitive recursive functions.

READ BJ page 92, line 4 to page 92 line -1.

The clearest way to set out the values of a and q is to use tables. For **BJ** Example 8.3, these tables are:

		y	
	$a(x,y)$	0	1
	1	1	3
x	2	0	2
	3	0	2
	4	0	1

$a(x,y) = y$ otherwise.

		y	
	$q(x,y)$	0	1
	1	2	1
x	2	0	3
	3	0	4
	4	0	0

$q(x,y) = 0$ otherwise.

The point is that the information contained in a flow graph is completely specified by the corresponding functions (or tables) a and q, and vice versa (just as the machine table or set of quadruples determine the same program as the corresponding flow graph). Thus, in the above example, the fact that $a(3,1) = 2$ and $q(3,1) = 4$ tells us that when this particular machine is in state q_3 and is scanning the symbol 1, then it moves left and goes into state q_4. This corresponds to part of the flow graph on **BJ** page 92. Notice that if $q(x,y) = 0$ (e.g. for $x = 3$, $y = 0$ in the example), then there is no instruction which the machine is to carry out when it is in state q_x and is scanning the symbol y, so the machine halts. This is what is meant by the sentence 'To halt, then, is to enter notional state 0' on **BJ** page 92, lines 12 and 13.

The fact that the functions a and q are always primitive recursive can be proved using exactly the same technique as used in Problem 7 of Section 4.1, where it was shown that the function a of the particular example above is primitive recursive.

PROBLEM 10 Write down the corresponding tables of values $a(x,y)$ and $q(x,y)$ for the Turing machines given by:

(i) the machine table in Problem 3 of Section 1.3 of *Unit 1*;

(ii) the flow graph on **BJ** page 28.

PROBLEM 11 Draw the flow graph of the Turing machine given by the following tables of values of $a(x,y)$ and $q(x,y)$:

		y	
	$a(x,y)$	0	1
	1	2	2
	2	3	2
x	3	1	0
	4	2	3
	5	0	1

$a(x,y) = y$ otherwise.

		y	
	$q(x,y)$	0	1
	1	1	2
	2	5	3
x	3	5	4
	4	4	3
	5	0	0

$q(x,y) = 0$ otherwise.

SOLUTION 10 (i)

		y	
	$a(x,y)$	0	1
	1	0	0
	2	1	0
x	3	1	0
	4	3	3
	5	3	2
	6	3	1

$a(x,y) = y$ otherwise.

		y	
	$q(x,y)$	0	1
	1	0	4
	2	6	5
x	3	4	6
	4	2	2
	5	3	3
	6	1	0

$q(x,y) = 0$ otherwise.

Note that e.g. $a(6,1) = 1$ and $q(6,1) = 0$ following these 'otherwise' rules, as there is no information given for what happens when $x = 6$, $y = 1$.

(ii)

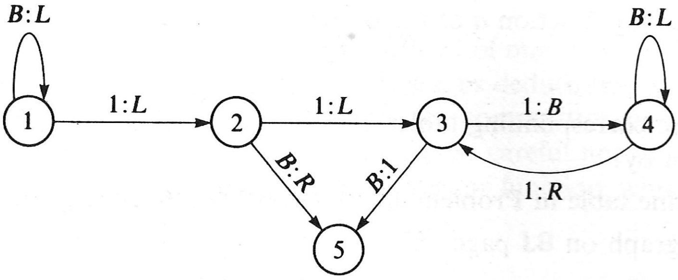
		y	
	$a(x, y)$	0	1
	1	0	0
	2	3	1
x	3	1	0
	4	3	1
	5	0	1

$a(x, y) = y$ otherwise.

		y	
	$q(x, y)$	0	1
	1	0	2
	2	3	0
x	3	5	4
	4	1	0
	5	0	0

$q(x, y) = 0$ otherwise. ■

OLUTION 11



■

4.3 Turing Computable Functions are Recursive

We are now ready to complete the proof that if f is a Turing computable function whose values are computed by the machine M then f is recursive.

Following **BJ**, we use the variable t to represent the various stages of M's computation. The configuration of M will depend both on the pair of numbers (x_1, x_2) initially on the tape as input (i.e. at stage 0), and on the stage t of the computation. Thus M's calculation is described by three functions, l, r and c, defined as follows:

$l(x_1, x_2, t)$ = the left number of M with initial input (x_1, x_2) at stage t;
$r(x_1, x_2, t)$ = the right number of M with initial input (x_1, x_2) at stage t;
$c(x_1, x_2, t)$ = the number of the state that M with initial input (x_1, x_2) is in at stage t.

Thus, for example, we saw in the last section that for any machine M we have

$$l(x_1, x_2, 0) = 0 \text{ and } r(x_1, x_2, 0) = s(x_1, x_2)$$
$$= (2^{(x_2+1)} \dot{-} 1) \cdot 2^{(x_1+2)} + (2^{(x_1+1)} \dot{-} 1)$$

(see **BJ** page 91, line -9); further, $c(x_1, x_2, 0) = 1$ since we suppose that M starts its computation in state q_1.

We wish to show that all these three functions are primitive recursive. Now if we were able to express, for example, $l(x_1, x_2, t+1)$ as a primitive recursive function of $l(x_1, x_2, t)$, we could indeed show that l was primitive recursive. However, you will see that $l(x_1, x_2, t+1)$ depends not only on $l(x_1, x_2, t)$ but also on $r(x_1, x_2, t)$ and $c(x_1, x_2, t)$. Thus l cannot be defined by a straightforward use of primitive recursion. The way out of this difficulty is to replace the three numbers which describe M's configuration (for given input at a given stage) by a single number which encodes all the information given by the three separate numbers. In this way, the three functions l, r, c are replaced by a single function.

READ BJ page 93, line 1 to page 93 line 21.

NOTES (i) *Page 93, line 13*
We showed tpl to be primitive recursive in Problem 17 of Section 3.1 of *Unit 3*.

(ii) *Page 93, line 18*
The reference to 7.19 here is to **BJ**'s proof that these functions are primitive recursive. Our proof appears in Solution 10 of Section 4.1. ▲

PROBLEM 1 Calculate $\text{tpl}(3, 2, 1)$.

PROBLEM 2 Which triple is encoded by the number 270,000?

SOLUTION 1 $\text{tpl}(3, 2, 1) = 2^3 \cdot 3^2 \cdot 5^1 = 8 \cdot 9 \cdot 5 = 360$. ■

SOLUTION 2 Factorizing 270,000 yields $270{,}000 = 2^4 \cdot 3^3 \cdot 5^4$. Hence
$270{,}000 = \text{tpl}(4, 3, 4) = \langle 4, 3, 4 \rangle$. ■

We are now able to define a single function g which completely describes the behaviour of the machine M, and we do this by giving g the rule

$$g(x_1, x_2, t) = \langle l(x_1, x_2, t), c(x_1, x_2, t), r(x_1, x_2, t) \rangle.$$

We now come to the crux of the proof, which is to show that this function g is primitive recursive. It will then be a relatively easy matter to deduce that f (the function computed by M) is recursive.

Before asking you to read the full primitive recursive definition of g given in **BJ**, which is a rather complicated definition by cases, we first give a rough idea and then work through one of the cases in detail.

We wish to express $g(x_1, x_2, t + 1)$ in terms of $g(x_1, x_2, t)$. Let us write

$$l = \text{lft}(g(x_1, x_2, t)), \quad c = \text{ctr}(g(x_1, x_2, t)), \quad r = \text{rgt}(g(x_1, x_2, t)),$$

and

$$l' = \text{lft}(g(x_1, x_2, t + 1)), \quad c' = \text{ctr}(g(x_1, x_2, t + 1), \quad r' = \text{rgt}(g(x_1, x_2, t + 1)).$$

So at stage t, the left number and right number of M's configuration are l and r, and M is in state number c; similarly, at stage $t + 1$, the left number and right number of M's configuration are l' and r', and M is in state number c'. It will be sufficient to express l', r', c' in terms of l, r, c, which amounts to describing the effect on l, r, c of the step in M's computation that takes the machine from stage t to stage $t + 1$.

Now the scanned symbol at stage t is the end digit of r, and hence will be 0 or 1 according as r is even or odd, i.e. the scanned symbol is $e(r)$, where e is the function defined by

$$e(x) = \begin{cases} 0, & \text{if } x \text{ is even,} \\ 1, & \text{if } x \text{ is odd.} \end{cases}$$

(We met this function in Problem 14 of Section 3.1 of *Unit 3*, where we proved it was primitive recursive.)

It follows that the state of M at stage $t + 1$ is $q(c, e(r))$ (by definition of the function q). Hence $c' = q(c, e(r))$.

We now use the results of Section 4.2 to determine l' and r'. These numbers will depend on what the machine does between stage t and stage $t + 1$ of the computation: the machine will do one of the actions 'write B', 'write 1', 'move left' or 'move right', corresponding respectively to $a(c, e(r)) = 0, 1, 2$ or 3 (by definition of the function a on **BJ** page 92). Let us consider the case $a(c, e(r)) = 2$, corresponding to the situation when M moves left. In Section 4.2, we saw that the new values l', r' of the left and right numbers in this case are given as follows:

$$\text{if } l \text{ is even, } l' = \frac{l}{2} \quad \text{and} \quad r' = 2r,$$

$$\text{if } l \text{ is odd, } l' = \frac{l-1}{2} \quad \text{and} \quad r' = 2r + 1.$$

Thus when $a(c, e(r)) = 2$ and $e(l) = 0$ the values of l', c', r' are given by

$$l' = \tfrac{1}{2}l, \quad c' = q(c, e(r)), \quad r' = 2r,$$

and hence $g(x_1, x_2, t + 1) = \langle \tfrac{1}{2}l, q(c, e(r)), 2r \rangle$.

Similarly, when $a(c, e(r)) = 2$ and $e(l) = 1$,

$$g(x_1, x_2, t + 1) = \left\langle \frac{l \dot{-} 1}{2}, q(c, e(r)), 2r + 1 \right\rangle.$$

It is now easy to see that these equations express $g(x_1, x_2, t + 1)$ as a primitive recursive function of $g(x_1, x_2, t)$. For example, writing out the first equation in full gives

$$\begin{aligned} &g(x_1, x_2, t + 1) \\ &= \mathrm{tpl}(\tfrac{1}{2} \cdot \mathrm{lft}(g(x_1, x_2, t)), q(\mathrm{ctr}(g(x_1, x_2, t)), e(\mathrm{rgt}(g(x_1, x_2, t)))), 2 \cdot \mathrm{rgt}(g(x_1, x_2, t))) \end{aligned}$$

and only compositions of known primitive recursive functions appear on the right-hand side. The term $\frac{1}{2} \cdot \mathrm{lft}(g(x_1, x_2, t))$ presents no problems, since we may replace it with $j(\mathrm{lft}(g(x_1, x_2, t)))$, where j is the function of Problem 9 of Section 4.1.

Of course, this equation holds only under the conditions $a(c, e(r)) = 2$ and $e(l) = 0$ (which can be shown to be a primitive recursive condition using the results of Section 4.1). There are seven other possibilities, which are specified in the next reading passage. We suggest you try working out the formula for the values of $g(x_1, x_2, t + 1)$ in other cases before reading the passage—the next problem contains one such case.

PROBLEM 3 Express $g(x_1, x_2, t + 1)$ in terms of l, c and r in the case $a(c, e(r)) = 1$, $e(r) = 0$.

SOLUTION 3 We must discover the effect on l, c and r when M changes the scanned symbol, which is 0 since $e(r) = 0$, to 1. As before, we have $c' = q(c, e(r))$, and by the results of Section 4.2 we have $l' = l$ and $r' = r + 1$. Thus

$$g(x_1, x_2, t + 1) = \langle l, q(c, e(r)), r + 1 \rangle. \quad \blacksquare$$

READ BJ page 93, line 22 to page 95, line 5.

NOTES (i) *Page 94, lines 14 to 16*
The function / is not needed. in fact, the function j of Problem 9 of Section 4.1 may be used instead as we need only division by 2.

(ii) *Page 94, lines 24 to 25*
We explain this further. Suppose M halts at stage t. Let $g(x_1, x_2, t) = \langle l, c, r \rangle$. Then, by the definition of g, $\mathrm{ctr}(g(x_1, x_2, t + 1)) = q(c, e(r))$. However, M has halted at stage t (in state number c), which is the same as saying that M has no actual state it can enter when in state c scanning the symbol $e(r)$. It was precisely in this situation that we defined $q(c, e(r)) = 0$.

(iii) *Page 94, lines −4 to −1*
Notice also that if $f(x_1, x_2)$ is undefined, then M never halts (with input (x_1, x_2)) so $\text{ctr}(g(x_1, x_2, t))$ is never 0. This implies that $\text{Mn}[h](x_1, x_2)$ is undefined, which in turn implies $\text{lo}(\text{rgt}(g(x_1, x_2, \text{Mn}[h](x_1, x_2))))$ is undefined.

(iv) *Page 95, lines 1 to 5*
The proof of Kleene's normal form theorem follows from the fact that all the functions lo, rgt, g, h appearing in the formula for f are primitive recursive and hence can be obtained without using minimization. Hence the only minimization used in this recursive definition of f is that applied to h, which is a *total* function (recall our definition of recursive function on page 60 of *Unit 3*). ▲

4.4 Church's Thesis

We have now completed the first half of this course, in which we have tried to make precise the intuitive notions of 'algorithm' and 'computable function'. Recall that Church's thesis states that the intuitively computable functions are exactly those which can be computed on a Turing machine. We can, of course, never give a precise proof of this (which is why it is called a thesis rather than a theorem), because its very statement involves human intuition which we can never (presumably) make precise. However, we can produce evidence for the thesis, which we now summarize in the light of the material we have covered in the last four units.

We hope that your experience with Turing machines has convinced you that their mode of operation *is* purely mechanical—that is, that any Turing computable function is computable in the intuitive sense. The problem lies in the fact that someone may produce a non-Turing-computable function together with a convincing algorithm that computes this function. This has certainly never been done, but there are better reasons for believing in Church's thesis than the current lack of a counterexample.

Firstly, the notion of a Turing computable function is very 'stable'. That is, it makes no difference if our definition of a Turing machine allows such variations as more than one tape, or a two-dimensional tape, or two or more scanning heads moving simultaneously etc. (That such machines are no more powerful is proved by imitating their actions on a one tape Turing machine in much the same way as we imitated the action of an abacus machine on a Turing machine.)

The most convincing evidence for Church's thesis, however, is that all attempts at making the notion of algorithm precise have resulted in the corresponding classes of computable functions being the same. We have proved this with three such notions in the preceeding units but many more 'machine models' have been (independently) invented and the result is always the same: it has always been possible to simulate the operations of such machines on a Turing machine. It seems implausible that everyone has made the same mistake!

We shall thus assume Church's thesis from now on when discussing the philosophical questions posed in the introduction to the course. In particular, recall that Leibniz's question in Section 1.0 of *Unit 1* asked whether there is an algorithm for deciding whether a given statement of number theory is true or false. We shall therefore take the word algorithm here to mean a process programmable on a Turing machine. The purpose of the next three units is to make the phrase 'statement of number theory' precise, so that we can use such statements as inputs on a Turing machine. The last unit of the course will then provide the answer to this now precise reformulation of the question.

4.5 Objectives of Unit 4

We list those things on which we may set assessment questions to test your understanding of the unit. After working through the unit, we feel you should be able to:

(i) decide whether a set of conditions on numbers or on pairs of numbers is mutually exclusive or collectively exhaustive;

(ii) show that a condition is primitive recursive, by showing its characteristic function is primitive recursive;

(iii) compute the values of a recursive function whose definition includes use of either definition by cases or general summation;

(iv) prove a function is primitive recursive by defining it in terms of given primitive recursive functions, by means of definition by cases or general summation (as well as the means described in *Unit 3*);

(v) understand the proofs that the functions d, lo, a, q, tpl are primitive recursive;

(vi) given the configuration of a Turing machine tape, compute its left and right numbers, and conversely, given these numbers, describe the configuration of the tape;

(vii) describe the effect on left and right numbers as the configuration of a Turing machine changes;

(viii) given a Turing machine, define the value of the corresponding functions a and q, and show that these functions are primitive recursive.

(Note we shall not assess the full proof that every Turing computable function is recursive.)

Index (Units 1 to 4)

Words

abacus, **BJ** 55
abacus computable, 51
absolute difference, 65
algorithm, 8
algorithmically computable, 8
argument, 7

basic function, **BJ** 74
binary numeral, 83
block diagram, 36, **BJ** 56
box, 68, **BJ** 55
B-representation, 54

cases, definition by, **BJ** 85
characteristic function, **BJ** 85
Church's thesis, 9, 91, **BJ** 20
codomain, 7
composition, **BJ** 75
computable function, 9
conditions, 76
  collectively exhaustive, 76
  mutually exclusive, 76
  primitive recursive, 77
configuration, **BJ** 23
countable, 7

denumerable, **BJ** 1
difference function, 39, 64
domain, 7

enumerable, **BJ** 1
enumerably infinite, **BJ** 1

flow chart, **BJ** 55–56
flow graph, 11, **BJ** 23
function, 7
  abacus computable, 51
  basic, **BJ** 74
  partial, 7
  primitive recursive, 64
  recursive, 72
  regular, 72
  total, 7
  Turing computable, 9

halting problem, **BJ** 41
Hilbert's question, 6

identity function, **BJ** 74
input, 14

Kleene normal form, **BJ** 95

left number, numeral, **BJ** 90
Leibniz's question, 6

machine table, 11, **BJ** 23
minimization, **BJ** 82
  regular, 72
monadic notation, 14
  for pairs of integers, 17

natural numbers, set of, 6

1-representation, 54
output, 14

pair function, 66
parity function, 17
partial function, 7
plugging together, **BJ** 37
predecessor function, 64
primitive recursion, **BJ** 76
primitive recursive
  condition, 77
  function, 64
product, general, 80
productivity, **BJ** 34
projection function, **BJ** 74

quadruple, 11, **BJ** 23

range, **BJ** 6
recursive function, 72
register, **BJ** 54
register table, 35
regular function, 72
regular minimization, 72
right number, numeral, **BJ** 90

satisfies, 76
signum, 38, 65
starting position, standard, 14
  for a pair (m, n), 17
state, **BJ** 21
string, finite, 7
successor function, 59, **BJ** 71
summation, general, 79

total function, 7
trace table, 35
triple function, 66, **BJ** 93
Turing computable function, 9
Turing machine, **BJ** 21–22

value, 7

zero function, **BJ** 74

Function Names and Special Symbols

$a(x, y)$ **BJ** 92
adf 65
C_1 and C_2 78
Cn **BJ** 75
ctr 80
$d(x, y)$ 79
dif 39, 64
$e(x)$ 65
Eq 77
exp **BJ** 70
$f : A \longrightarrow B$ 7
$x \longmapsto f(x)$ 7
fac **BJ** 78
id **BJ** 74
id_i^n **BJ** 74
l 84
lft 80
lo 80
$[m] + [n] \longrightarrow p$ **BJ** 54
Mn$[f]$ **BJ** 82
N 6
$[n]$ **BJ** 54
$n+, n-$ **BJ** 56
not C 77
$p(n)$ **BJ** 35
P 7
$\prod_{z=0}^{y}$ 80
par 17
pr 66
Pr **BJ** 76
pred 64
prod **BJ** 76
$q(x, y)$ **BJ** 92
q_i 11, **BJ** 21
r 84
rgt 81
s 59, **BJ** 71
$s(x_1, x_2)$ **BJ** 91
$\sum_{n<x}$ 65
$\sum_{z=0}^{y}$ 79
$S_0, S_1, \ldots$ 11, **BJ** 21
sg 38, 65
$\overline{\text{sg}}$ 65
sum **BJ** 76
tpl 66, **BJ** 93
$z(x)$ **BJ** 74
$\dot{-}$ 39
$\langle x, y, z \rangle$ **BJ** 93
$\Longleftrightarrow$ 44

0 335 14012 2

COMPUTABILITY AND LOGIC

Third edition

George S. Boolos and
Richard C. Jeffrey

Open University Set Book

CAMBRIDGE
UNIVERSITY PRESS

This third edition of *Computability and Logic* has been corrected and contains thoroughly revised versions of the chapters on Ramsey and provability, with new exercises provided for three other chapters. There are also two new chapters dealing with undecidable sentences and on the non-existence of non-standard recursive models of *Z.*

". . . gives an excellent coverage of the fundamental theoretical results about logic involving computability, undecidability, axiomatization, definability, incompleteness, etc."

American Math Monthly

". . . particularly appropriate for graduate and advanced undergraduate students in philosophy. . . . The book is written in a clear and pleasing style and avoids pedantry. . . . It should be an excellent text for its intended audience."

Mathematical Reviews

CAMBRIDGE
UNIVERSITY PRESS

ISBN 0-521-38923-2

9 780521 389235